Aalto University
School of Science
Degree Programme of Computer, Communication and Information Sciences

Astaroth: A Library for Stencil Computations on Graphics Processing Units

Johannes Pekkilä

Master's Thesis
Espoo, May 20, 2019

Supervisors: Professor Petteri Kaski
Instructor: Matthias Rheinhardt, Ph.D.
Professor Jaakko Lehtinen

Aalto University
School of Science
Degree Programme of Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Johannes Pekkilä

Title:
Astaroth: A Library for Stencil Computations on Graphics Processing Units

Date: May 20, 2019 **Pages:** 111

Professorship: Computer Science **Code:** SCI3042

Supervisors: Professor Petteri Kaski

Instructor: Matthias Rheinhardt, Ph.D.
Professor Jaakko Lehtinen

Graphics processing units (GPUs) are coprocessors, which offer higher throughput and better power efficiency than central processing units in data-parallel tasks. For this reason, graphics processors provide a good platform for high-performance computing. However, programming GPUs such that all the available performance is utilized requires in-depth knowledge of the architecture of the hardware. Additionally, the problem of high-order stencil computations on GPUs in challenging multiphysics applications has not been adequately explored in previous work. In this thesis, we address these issues by presenting a library, an efficient algorithm and a domain-specific language for solving stencil computations within a structured grid. We tested our implementation by simulating magnetohydrodynamics, which involved the computation of first, second, and cross partial derivatives using second-, fourth-, sixth-, and eight-order finite differences with single and double precision. The running time of our integration kernel was 2.8–9.1 times slower than the theoretical minimum time, which it would take to read the computational domain and write it back to device memory exactly once, without taking into account the effects of finite caches or arithmetic operations on performance. Additionally, we made a performance comparison with a CPU solver widely used for scientific computations, which we benchmarked on a total of 24 cores of two Intel Xeon E5-2690 v3 processors. Our solver, benchmarked on a Tesla P100 PCIe GPU, outperformed the CPU solver by factors of 6.7 and 10.4 when using single and double precision, respectively.

Keywords: high-performance computing, stencil computations, graphics processing units, magnetohydrodynamics

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan tutkinto-ohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Johannes Pekkilä		
Työn nimi:	Astaroth: Ohjelmistokirjasto stensiililaskentaan grafiikkasuorittimilla		
Päiväys:	20. toukokuuta 2019	**Sivumäärä:**	111
Professuuri:	Tietotekniikka	**Koodi:**	SCI3042
Valvojat:	Professori Petteri Kaski		
Ohjaaja:	Matthias Rheinhardt, FT Professori Jaakko Lehtinen		

Grafiikkasuorittimet ovat apusuorittimia, jotka tarjoavat rinnakkain laskettavissa tehtävissä parempaa suoritus- ja energiatehokkuutta kuin keskussuorittimet. Tästä syystä grafiikkasuorittimet tarjoavat hyvän alustan suurteholaskennan tarpeisiin. Toisaalta grafiikkasuorittimen ohjelmointi siten, että kaikki tarjolla oleva suorituskyky saadaan hyödynnettyä vaatii syvällistä asiantuntemusta ohjelmoitavan laitteiston arkkitehtuurista. Korkean asteen stensiililaskentaa haastavissa fysiikkasovelluksissa ei ole myöskään tutkittu laajalti aiemmissa julkaisuissa. Tässä työssä otamme kantaa näihin ongelmiin esittelemällä ohjelmistokirjaston, tehokkaan algoritmin, sekä tehtävään räätälöidyn ohjelmointikielen stensiililaskujen ratkaisemiseen säännöllisessä hilassa. Testasimme toteutustamme simuloimalla magnetohydrodynamiikkaa, johon kuului ensimmäisen ja toisen kertaluvun derivaattojen lisäksi ristiderivaattojen ratkaisu toisen, neljännen, kuudennen ja kahdeksannen kertaluvun differenssimenetelmällä käyttäen sekä 32- että 64-bittisiä liukulukuja. Integrointifunktiomme suoritusaika oli 2.8–9.1 kertaa hitaampi kuin teoreettinen vähimmäisajoaika, joka menisi laskennallisen alueen lukemiseen ja kirjoittamiseen apusuorittimen muistista täsmälleen kerran, ottamatta huomioon äärellisen välimuistin tai laskennan vaikutusta suoritusaikaan. Vertasimme kirjastomme suoritusaikaa laajalti tieteellisessä laskennassa käytettyyn keskussuorittimille tarkoitettuun ratkaisijaan, jonka ajoimme kokonaisuudessaan 24:llä ytimellä kahdella Intel Xeon E5-2690 v3 -suorittimella. Tähän ratkaisijaan verrattuna Tesla P100 PCIe -grafiikkasuorittimella ajettu ratkaisijamme oli 6.7 ja 10.4 kertaa nopeampi 32- ja 64-bittisillä liukulukuvuilla laskettaessa, tässä järjestyksessä.

Asiasanat:	suurteholaskenta, stensiililaskenta, grafiikkasuorittimet, magnetohydrodynamiikka
Kieli:	Englanti

Acknowledgments

First of all, I would like to thank Maarit Käpylä for making the initial leap of faith and letting me into the world of academic research, for believing in me and for continuous support for all these years. I would also like to thank Matthias Rheinhardt for not sparing the red marker and for providing an endless stream of good questions, to which I thought I already knew the answer to, but realized that in reality I did not. I would like to thank my supervisor Prof. Petteri Kaski for patience, valuable advice and for driving me to exceed my own expectations. I would also like to thank Miikka Väisälä whom I have had the pleasure to work with on previous versions of Astaroth and who has not been afraid to take high-risk, high-reward decisions when it comes to academic research. I also thank Prof. Jaakko Lehtinen for valuable comments and suggestions. Finally, I would like to thank my family and friends for providing food, good company, and helping me unwind after long days working on this thesis.

I thank CSC IT Center for Science Ltd. for the compute resources used for this work. I acknowledge the financial support by the Academy of Finland to the ReSoLVE Centre of Excellence (project no. 307411).

Espoo, May 20, 2019

Johannes Pekkilä

Contents

Chapter 1

Introduction

Efficient computational methods are indispensable in computational sciences and high-performance computing, where a single line of code can be a matter of quadrupling the electric bill and time spent on waiting for the computation to complete. In physics, there is generally no limit to the performance improvement that would not be beneficial in some way. With grid-based fluid simulations for example, the resolution of simulation models could be increased virtually indefinitely in order to achieve better accuracy, or longer time periods could be simulated in a reasonable time.

Until 2003, computational scientists were content with uniprocessor performance growing at a rate of 52% per year, as simulation times were expected to halve every 18 months just by acquiring new hardware and relying on compilers to finish the job [1–4]. However, the free lunch is said to be over [5] as microprocessor manufacturers have hit a power limit, which makes it infeasible to dissipate excess heat from mass-marketable microprocessors running at higher clock rates [6]. The power wall has been the driving force behind the shift from single-core to many-core architectures. As a result, performance-critical programs must now be written to take advantage of multiple processors.

While modern central processing units (CPUs) generally house from 4 to 64 cores, graphics processing units (GPUs) can house a few thousand functional units capable of executing arithmetic instructions simultaneously. Originally designed for high throughput in real-time rendering, GPUs have the potential to accelerate data-parallel tasks by an order of magnitude over a CPU[1]. As modern GPUs can also be programmed using a general-purpose language, such as CUDA [9] and

[1]Comparing the theoretical maximum memory bandwidth and floating-point operations per second of a Xeon E5-2690v3 12C [7] and a Tesla P100 [8].

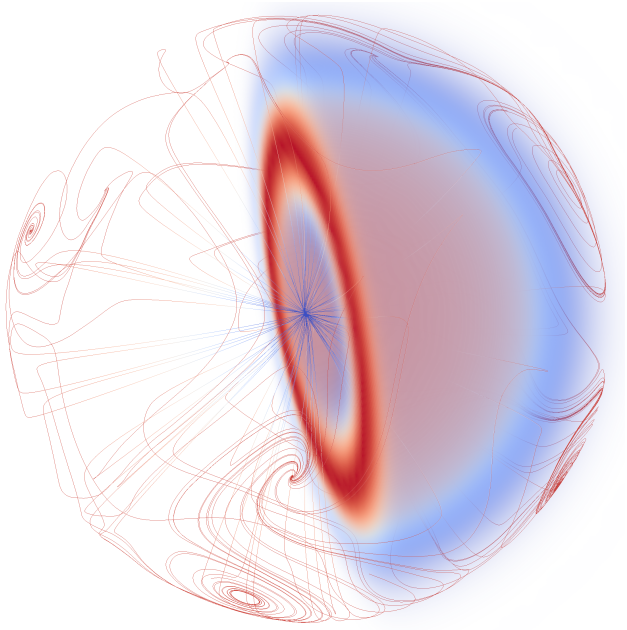


Figure 1.1: Compressible non-isotermal flow undergoing a forced radial explosion.

OpenCL [10], this makes GPUs an attractive platform for solving many tasks found in computational sciences.

Our motivation behind this work is to simulate the flow of compressible, electrically conducting fluids. The study of such fluids is called magnetohydrodynamics. Figure 1.1 shows a simulation of a compressible fluid, where a turbulent flow is undergoing a radial explosion. Throughout this work, we focus on the computational aspects of such simulations and refer the reader to [11, 12] for a detailed discussion on the physical properties of magnetohydrodynamics. Common to all multiphysics simulations, magnetohydrodynamics involves multiple fields, which may interact with each other [13]. We refer to such fields as being *coupled*. However, literature on accelerating simulations involving a large number of coupled fields is limited, as the majority of previous work focuses on idealized test cases, which do not capture the difficulties arising when implementing complex multiphysics solvers on multi-core processors. In this work, we explore those issues and implement a parallelized solver for magnetohydrodynamics.

Central to this work is the finite-difference method, which is commonly used for discretizing continuous fields to approximate solutions to partial differential equations [14]. With finite differences, derivatives at each discrete point of a field can be approximated based on the values of its neighboring points. Functions, which update an array of elements by computing derivatives in this fashion belong to a class of iterative algorithms called stencil codes [15]. In this case, a stencil

consists of the neighboring points required to compute the derivative. Because stencil computations are used also in other problem domains [13, 16], the contributions of this work are more useful to the scientific community if we do not limit ourselves to fluid simulations. Therefore our focus in this work is to accelerate stencil codes on GPUs, and use magnetohydrodynamics as a test case to evaluate whether our implementation is also suitable for challenging multiphysics applications.

There is also a disconnect between the needs of computational scientists and the tools available for implementing efficient parallel programs [4, 17, 18]. While the focus of computational scientists is to develop new mathematical and physical models, using the available tools requires expertise on the execution model and features of the hardware the programs are run on. Therefore it would be highly beneficial, if a subset of problems could be expressed with a high-level language, which captures only the details required to implement the model, while tuning the code could be done with automated tools [4].

The issues in optimizing complex multiphysics simulations and the lack of high-level tools lead to our research questions. First, is there an efficient algorithm for solving high-order stencil computations on a GPU in a three-dimensional grid, where each grid point is connected to the same number of neighbors and the computations involve multiple coupled fields? Second, can such stencil computations be expressed with a programming language, that captures a high-level representation of a numerical model, but which can also be translated into efficient, parallelized code?

In this work, we present a suite for processing three-dimensional stencils efficiently on graphics processing units. The suite consists of three subcontributions. First, we present a new library and an application programming interface for managing the resources of GPUs in a compute node. Second, we present a novel domain-specific language, which can be used to express computations in a grid using a high-level representation. Finally, we present a compiler, which can translate programs written in our domain-specific language into efficient GPU kernels usable with the library. We evaluate our suite by implementing a magnetohydrodynamics solver with the domain-specific language and compare the performance of our solver with both the theoretical limits of the hardware and a mature CPU solver used in numerous publications, the Pencil Code [19].

1.1 Background

Stencil codes are a class of iterative functions, where neighboring data elements of are used to generate a new value [15]. Stencil computations are encountered in a multitude of applications in science and engineering [4, 13, 14], from image processing [20, 21] to solving partial differential equations, such as simulations of hydrodynamics [22] and astrophysical gases [11]. In 2004, Colella [16] identified stencil computations as a class of numerical methods, which he expected to be relevant in computing for the next ten years. Colella's list of the vital numerical methods was later revisited and extended by Asanovic *et al*. [4, 16], who argued that instead of relying on improvements in single-core performance, the focus in computer sciences should steer towards finding ways to express parallel problems with a high-level representation, and rely on automated tuning to produce efficient code especially with these methods. At the time Colella and Asanovic *et al*. made their observations, the yearly improvement rate in single-core performance had started to diminish [2], which signaled the start of significant development efforts to parallelize solvers in hopes to continue to have performance gains with future hardware.

The fundamental shift from single-core to many-core architectures was initiated after microprocessor manufacturers hit three walls in improving single-core performance [2–4, 16, 23]. First, the memory wall is characterized by the disproportional increase in arithmetic performance to the number of bytes that can be serviced by the memory system [23]. This wall is encountered when a problem becomes inherently bound by the main memory bandwidth when the operational intensity of a kernel is not high enough to saturate the available arithmetic-logic units. Because the operational performance of microprocessors increases at a faster rate relative to memory bandwidth, this issue is accentuated with every hardware generation. Second, the power wall limits the highest clock rate that can be achieved before cooling the microprocessor becomes too expensive with mass-producible technology [6]. Increasing the clock rate and lowering the voltage indefinitely is infeasible, as powering a large number of densely packed transistors, or running the system at lower voltages increases energy dissipation [3]. The third wall is caused by diminishing returns when attempting to improve performance by utilizing deeper instruction pipelines [2]. With instruction pipelines, hardware designers seek to exploit instruction-level parallelism, which is a form of parallelism where multiple instructions are executed in parallel [3]. Instruction-level parallelism with instruc-

tion pipelines is usually achieved by executing an instruction fetch-decode-execute cycle in multiple steps, such that the steps can be executed in parallel and the next instruction can be issued before the previous instruction has finished [2, 3]. An alternative approach is to issue multiple instructions to multiple functional units during a single clock cycle [2, 3]. Registers and caches are limited, and branches cannot be predicted perfectly, which imposes a limit on the performance that can be achieved with instruction-level parallelism [2].

Central processing units have traditionally been used for high-performance computing. They are designed to perform well in general-purpose tasks, where concurrently executed programs may access the resources of the hardware relatively efficiently. To this end, CPU cores are deeply pipelined and instructions are issued depending on whether the data required by the instruction is available. Modern CPUs execute instructions out-of-order, as the order in which the instructions are executed is not necessarily the original ordering defined in the program [2, 24]. For example, an execution core of a Sandy Bridge CPU has six dispatch ports, which can issue six microinstructions to the functional units of the core each clock cycle [24]. In such a superscalar pipeline, a core might be capable of encoding an instruction, executing an arithmetic operation and servicing a memory transaction during a single clock cycle. A functional unit may comprise for example and arithmetic-logic unit used to compute arithmetic operations, or a load-store unit used for executing instructions to read and write from memory locations. Because cores execute instructions in a deeply pipelined fashion, mispredicting a branch can be very costly because the pipeline has to be filled with instructions and data might have not been prefetched into caches [2]. For this reason, CPU core also house branch predictors, which predict the branch based evaluations of previous branches [2]. In order to mitigate the penalty of cache misses and to reduce pressure to main memory, CPUs employ large multilevel caches, and can achieve very high cache hit rates with well-written programs [3].

In the 1990s, GPUs emerged as highly parallel coprocessors designed for high throughput in graphics-related tasks [25]. To this end, GPUs specialize in data-parallel problems, where a single instruction can be executed on multiple data items in parallel. With modern devices, each instruction is executed on 32–64 stream processors at a time [9, 10]. A stream processor is analogous to a functional unit of a CPU executing arithmetic instructions. For example, a Tesla P100 PCIe GPU contains a total of 3584 stream processors [8]. Additionally, GPUs utilize a specialized memory setup, which enables an order of magnitude higher bandwidth than what is

achievable with memory systems used with CPUs [2]. However, the higher throughput of the memory system comes with the cost of longer access latency [25]. With GPUs, the long latency of memory accesses has been mitigated by adopting an execution model, where a large number of threads are run on a processor in a fine-grained fashion instead of relying on large multilevel caches to avoid accessing the main memory [3]. With fine-grained multithreading, the thread being executed is switched out after each instruction to another thread that is ready to run [3, 25]. As long as there is at least one non-stalling thread capable of executing instructions, the execution units of the device are busy and the latency of memory operations is said to be hidden.

By the early 2000s, GPUs could be programmed by writing vertex and fragment shaders using assembly and higher-level shading languages. In the following five years, GPUs evolved from a rigid fixed-function pipeline into a more generic processor, where all programmable stages were based on a common virtual machine, which provided some of the arithmetic and logic capabilities previously found only on a CPU [27, 28]. Since then, there has been emergence of multiple application programming interfaces (APIs) for general-purpose computing on GPUs, namely OpenCL [10], OpenACC [29] and NVIDIA's CUDA [9]. The latest graphics APIs, Metal [30], Direct3D [27], OpenGL [31], and Vulkan [32] also offer compute shaders as a way to to express general computations without having to resort to using graphics primitives. These APIs offer a varying level of abstraction of the graphics pipeline.

However, creating software to make use of these coprocessors takes a significant investment in research and development [18, 33, 34]. In many cases, achieving the best possible performance with any of these languages requires expert knowledge of the hardware [17, 35]. While many traditional optimization strategies apply with GPUs, there are also key differences in the architecture, which must be taken into account. This in turn, is a hindrance to mathematicians and physicists, whose primary focus is to develop new mathematical and physical models. Higher-level programming paradigms exists, such as OpenACC [29], which work efficiently in the most common use cases, but for more complex problems are argued to lack the expressivity required to translate the program into efficient machine code [18, 36, 37].

In computational sciences, a programming language, which is easy to read and write, but also uses the resources of any piece of hardware to its fullest would be highly desirable. One way to solve the dilemma between generality and performance is to use a domain-specific language. In contrast to general-purpose languages such

[2] Intel Xeon E5-2690 v3 CPU provides a total of 64 GiB/s memory bandwidth [7], while a GP100GL Tesla P100 PCIe GPU can supply data at a rate of 682 GiB/s from device memory [26].

as C and Fortran, domain-specific languages are tailored towards solving a specific class of problems. By limiting the problem domain, the syntax of a domain-specific language can be much simpler than what is required from general-purpose languages, while the compiler for the domain-specific language can be tuned to generate highly efficient lower-level code, that exhibits performance close to handcrafted implementations in that domain [17, 18]. Listings C.2 and 1.2 showcase the differences between general-purpose and domain-specific languages in stencil computations.

```
template <int step_number>
static __global__ void
__launch_bounds__(RK_THREADBLOCK_SIZE, RK_LAUNCH_BOUND_MIN_BLOCKS)
solve(const int3 start, const int3 end, const float dt,
      VertexBufferArray buffer)
{
    const int i = threadIdx.x + blockIdx.x * blockDim.x + start.x;
    const int j = threadIdx.y + blockIdx.y * blockDim.y + start.y;
    const int k = threadIdx.z + blockIdx.z * blockDim.z + start.z;
    const int idx = i + j * MX + k * MX * MY;

    if (i >= end.x || j >= end.y || k >= end.z)
            return;

    const PreprocessedVertex3 T = stencil_assembly(i, j, k,
                                                   buffer.in[0],
                                                   buffer.in[1],
                                                   buffer.in[2]);
    const float3 result = heat_equation(T);

    #pragma unroll
    for (int w = 0; w < 3; ++w)
        buffer.out[w][idx] = buffer.in[w][idx] + result[w] * dt;
}
```

Listing 1.1: An example of optimized CUDA code.

```
in  Vector T     = (int3){0, 1, 2};
out Vector T_out = (int3){0, 1, 2};

Kernel
solve(Scalar dt)
{
    T_out = T + heat_equation(T) * dt;
}
```

Listing 1.2: An example of the domain-specific language developed for this work.

1.2 Previous work

There have been significant efforts to accelerate stencil processing on GPUs [17, 18, 20, 21, 38–43]. A majority of previous work has focused on optimizing computations with axis-aligned stencils ranging from 7 to 25 points. The optimizations generally

exploit the fact that the stencils used to update neighboring grid points overlap partially, and a carefully selected block of data can be cached and used for updating multiple neighboring grid points. This optimization technique is called cache blocking, where a function operates on blocks of data at a time and the blocks are small enough to fit into caches. Cache blocking is also commonly used for optimizing CPU programs, however, with GPUs the caches usually have to be managed by the programmer instead of relying on implicit caching. We discuss the features of GPU caches in detail in Chapter 2. A common approach to optimizing computations with axis-aligned stencils is to use 2.5-dimensional cache blocking [38, 39], where a two-dimensional slice of the data is fetched to the user user-managed cache and stencil points in the third dimension are implicitly stored into registers. With axis-aligned stencils, this approach significantly reduces traffic to off-chip memory when multiple points along the third axis are solved sequentially, as almost all the data required to update the next vertex is already resident in caches and registers. However, this approach is not suited for optimizing computations with complex stencils in high-order accurate simulations because of the limited size of the caches and register file on GPUs [44].

High-order accurate methods are often used in scientific computations, such as simulations of astrophysical gases [11, 19]. For example, a function computing first- and second-order derivatives in addition to cross partial derivatives may access data from 55 neighboring vertices. Such stencil is visualized in Figure 5.13. In addition, there may be a large number of coupled fields. In order to minimize redundant accesses to off-chip memory, a block of data containing data from all of the coupled fields should be resident in the cache when performing stencil computations with neighboring points. However, allocating a large amount of the cache for processing a group of stencils reduces the total number of threads that can be multithreaded on a GPU, which makes the program susceptible to instruction stalls due to the long latency of memory accesses. In high-order accurate simulations, finding the optimal balance between the portion of the cache allocated for reusing data and the number of threads being multithreaded on a GPU is a significant challenge that has not been addressed adequately in previous work.

As multiple fields are coupled in multiphysics applications, such as velocity and magnetic fields in magnetohydrodynamics, in terms of redundant memory transactions, the most efficient solution would be to update the system in a single pass, as in this case all the data required for updating a block of grid points could potentially be cached. To our knowledge, no previous work exists on accelerating magnetohydrodynamics with high-order finite differences efficiently in a single pass. The

drawback of updating the system in a single pass is that each thread is likely to require more resources, which in turn reduces the number of threads that can be multithreaded on the GPU. An alternative approach is to decompose the kernel and update the system in multiple passes, where each kernel requires less resources compared with a single-pass approach. In previous work, we reordered the computations in a way, which allowed us to update the system in two passes using axis-aligned stencils [44]. However, as data in the caches of a GPU are not coherent across kernel launches, updating the system in multiple passes required the reading and writing of intermediate values redundantly from and to off-chip memory.

General-purpose computing on GPUs is still a relatively new field, which is advancing at a rapid rate. There are a wide variety of libraries for solving popular tasks on GPUs, such as Polymage [20] and Halide [21] for image processing, and Torch [45] and TensorFlow [46] for machine learning. Work on high-order accurate multiphysics solvers is more limited. Magnetohydrodynamics solvers capable of utilizing GPUs include GAMER-2 [47] and ENZO [48]. These solvers use the piecewise parabolic method, which is related to the finite volume method. GAMER-2 and ENZO use adaptive mesh refinement, where the mesh is partially subdivided to provide finer resolution in areas where more accuracy is required, such as small-scale perturbations [47–49]. Magnetohydrodynamics solver closest to this work is Fargo3D [50], which uses finite differences for solving some differential equations, and the finite volume method for some. Fargo3D uses a parser to generate CUDA kernels from loops written in C, which conform to a specific format. The generated code is a straightforward conversion from C to CUDA and no significant optimizations are applied during the translation process. GAMER-2 and ENZO utilize shared memory at least in part of the computations [47, 48] and SBLOCK uses an approach similar to 2.5-dimensional cache blocking [40] while Fargo3D relies on implicit caching and finds the optimal problem decomposition with an auto-tuning script [50]. Comparison of related work closest to ours is shown in Table 1.1

There has been extensive work on auto-tuning generators of stencil kernels on GPUs, starting form Datta *et al*. [39] to others [42, 43]. However, the optimization techniques used in these types of studies are often too limited for multiphysics applications, as they focus on idealized test cases and do not address the problem with efficient caching when performing computations with large stencils and multiple coupled fields.

More general frameworks focused on solving partial differential equations in structured grids on GPUs include SBLOCK [40] and Cactus [51, 54]. Usability of these projects have been demonstrated in hydrodynamics applications. Cactus provides

Table 1.1: An overview of the previous work closest to ours. If the listed project was evaluated in multiple test cases, we included the test case closest to our test case. The methods used in these publications were the finite-difference method (FDM), the finite-volume method (FVM), the piecewise parabolic method (PPM) and adaptive mesh refinement (AMR). The details are as accurate as is stated in the cited publications. Incomplete data has been left out. For example, the accuracy of the method used was not explicitly stated in the publications of Cactus [51] or Delite [18].

Project	**Method**	**Test case**
Astaroth	6th-order FDM	MHD
Pencil Code [19]	6th-order FDM	MHD
Fargo3D [50]	Hybrid FDM and FVM	MHD
ENZO [48]	AMR	MHD
Cactus [51]	FDM	Hydrodynamics
PEnGUIn [52]	PPM	Hydrodynamics
GAMER-2 [47]	partial AMR	Hydrodynamics
SBLOCK [40]	4th-order FVM	Hydrodynamics
Astaroth Code [44]	6th-order FDM	Hydrodynamics
Lift [17, 53]	2nd-order FDM	Acoustics
Delite [18]	Not specified	Shallow water

abstractions for multiple tasks, such as data allocation, which allows the user to write platform-independent components on top of the framework using Fortran, C, or C++. SBLOCK parses python scripts into CUDA code in a similar fashion as Fargo3D, but with more focus on the optimization of the generated code. However, the optimization strategies used by SBLOCK are based on work considering 7-point stencils with no couplings [39, 40], which is why we suspect the approach would not work well with larger stencils in multiphysics applications.

So far we have discussed hand-coded physics solvers and libraries for stencil processing, which may rely on parsing scripts to generate kernels from code following a strictly defined format. However, parsing scripts may only be used for a straightforward translation of the code and more advanced tools are required to apply subtler optimizations. In recent years, there have been an emergence of projects utilizing optimizing compilers, which generate efficient code by analyzing a high-level domain-specific language [20, 21, 55]. In this work, we have drawn inspiration from PolyMage [20] and Halide [21]. These projects supply a novel language and a compiler for creating efficient image processing pipelines. The compilers in these projects decouple the execution order of subroutines from the high level description of the algorithm. PolyMage uses polyhedral analysis to reschedule subroutines,

while Halide chooses from a set of choices. The goal is, that the subroutines and their dependencies are discovered and reordered in a way that attempts to find a balance between redundant computation, redundant memory fetches and resource usage. For example, reduction from a vector to a scalar may be precomputed, but this requires that the result is held in caches until the result is no longer needed instead of fetching it redundantly from memory each time [21].

However, PolyMage and Halide are designed for two-dimensional image processing and are not stated to support double precision. To the best of our knowledge, no analogous projects for computations with larger three-dimensional stencils exist. In this work, we use a similar, however much simpler, approach to translate a domain-specific language to a stencil pipeline, which works efficiently with large three-dimensional stencils and multiple coupled fields.

The drawback of creating a new domain-specific language and an accompanying compiler suite is, that it requires expert knowledge in various fields, at least in compilers, computer architecture, and the methods to be used with the domain-specific language. In addition, maintaining a compiler requires a significant amount of work if the goal is to generate efficient code for multiple architectures. To our knowledge, this issue has been explored in two major projects so far, Delite [18] and Lift [17]. These compiler suites provide an intermediate language upon which higher-level domain-specific languages can be built, such as Liszt [56] designed for solving partial-differential equations. The benefit of an intermediate language is, that the designers of domain-specific languages do not have to create and maintain entire compilers, but can rather target an intermediate language which is then compiled further into optimized cross-platform code. It has been suggested that Delite and Lift are capable of producing code that exhibits performance close to hand-tuned implementations. Steuwer has reported that in two-dimensional stencil computations of unspecified order, Lift achieves 75% of the performance of a manually optimized implementation [57]. Sujeeth *et al*. have reported that several machine learning algorithms from K-means to linear regression generated with Delite achieve 85% or higher performance compared with hand-tuned implementations [18].

In this work, we adopt an approach closer to PolyMage and Halide, and create a new compiler, as this gives us more freedom to experiment with different optimization strategies. This is necessary, as computations with multiple coupled fields and large stencils have not been rigorously explored in previous work, which is why there is no guarantee that premade solutions for related, but simpler stencil computations would work well in our case. At a later date, our compiler suite can be modified to translate code into intermediate language for one of these projects if deemed useful.

The library introduced in this thesis, Astaroth, is a new software library based on the lessons learned during the development of its predecessor, the Astaroth Code [58]. The Astaroth Code is a proof-of-concept solver for simulating hydrodynamics, which we used in previous work to demonstrate that GPUs can be utilized efficiently for solving sixth-order stencil computations with 19- and 55-point stencils [44].

1.3 Problem statement

Our goal is to create a library tailored for the requirements of common multiphysics applications. In this section, we define those requirements. We use the word *must* for requirements, that need absolutely be fulfilled. Word *should* is used for requirements, that should preferably be fulfilled, but may be ignored given valid reasons. Finally, we use the word *may* to indicate requirements, that are optional.

First, magnetohydrodynamics equations solved using high-order finite differences described in Chapter 3 must be expressible using the domain-specific language created for this work. The solver must support both single- and double-precision arithmetic, and the number of fields that may be used in the calculations must not be fixed.

Second, the generated integration kernel for the specified test case should achieve performance within order of magnitude of the theoretical hardware maximum, calculated using a well-justified metric. Depending on the operational intensity of the problem, we consider a valid metric to be either the ratio of achieved arithmetic performance to the theoretical maximum utilization of the compute units, or the running time of the kernel to the minimum achievable time when only the memory bandwidth of the device is taken into account, assuming only the absolute minimum number of device memory transactions are serviced, and perfect caches and free arithmetic.

Third, the domain-specific language should be generic enough, such that it can be used to perform computations with arbitrary-sized stencils in a structured grid. In this work, we demonstrate that the language is generic enough to compute partial-differential equations with finite differences with multiple coupled fields in problems, which require a high order of accuracy. While this encompasses many problems found in multiphysics, we acknowledge that this is not an exhaustive test of all problems that can be solved with structured grids.

1.4 Outline of this work

The rest of this work is structured as follows. In Chapter 2, we introduce graphics processing units and cover the necessary details needed to understand the discussion on our implementations in later sections. In Section 2.1, we describe the execution model and architecture of graphics processing units, focusing on features used in general-purpose computing. We introduce concepts such as instruction-level parallelism, multithreading and latency hiding, which are crucial for achieving satisfactory performance with the execution model. We also give a short overview of the CUDA programming model and the compilation stages from CUDA to assembly-level code, and point out the challenges in utilizing the relatively small caches to avoid traffic to slower off-chip memory in Section 2.2.

In Chapter 3, we introduce the theoretical background of simulating compressible, electrically conducting fluids and describe the equations used to update the state of the simulation. In Section 3.1, we review the basics of the finite-difference methods, which we use to discretize the simulation domain spatially on a structured grid. We show how finite differences can be used to approximate derivatives at some point by using data from neighboring points. We also demonstrate, which neighbors are needed to update a point using the equations introduced earlier in this chapter. Finally in Section 3.2, we review the integration method which we use to advance the simulation in the time domain. We detail a low-storage Runge-Kutta scheme introduced by Williamson [59], and prove how the scheme can be rewritten in a form which is more suited for computations bound by memory bandwidth.

In Chapter 4, we present the architecture of our library, the grammar of our domain-specific language and how source files written in our language are compiled into GPU kernels. In Section 4.1, we detail the components of our library and the interface that can be used to control the execution on GPUs. In Section 4.2, we review the basics of compilers and context-free grammars, discuss the syntax of our domain-specific language and give an example of how the language can be used to solve a simple physical problem. Finally in Section 4.3, we describe the architecture of our compiler, detail the compilation phases from source code of our language to CUDA kernels and discuss the motivations, technical details and challenges in generating efficient kernels for solving computations with variable-sized stencils. In this section, we argue that implicit caching can give good results with complex stencil computations if the stencil operations are precomputed, the result is stored into caches and registers, and then used during later stages.

In Chapter 5, we present our results and evaluate the library in three tests. First in Section 5.1, we discuss the challenges in comparing floating-point numbers and verify that the results of our GPU solver are sufficiently close to a model solution. Second in Section 5.2, we describe the theoretical lower bound for the running time of the integration kernel, compare the performance of the kernel generated with our compiler with the lower bound, and show the detailed performance metrics of the kernel. We show that our integration kernel achieved roughly 18% efficiency when compared with a conservative performance bound when simulating compressible fluids using sixth-order finite differences to approximate derivatives. In Section 5.3, we compare the performance of our solver with the Pencil Code [19], which is a mature CPU solver used in a multitude of publications.

In Chapter 6, we explain the results and discuss the limitations of our tests. We also argue that aggressive use of caches is likely necessary to obtain better performance in bandwidth-bound problems, even if higher resource usage reduces the number of threads that can be multithreaded on the GPU. Finally, we contrast our results with previous work, discuss future work and conclude the thesis.

Chapter 2

Graphics processing units

In this chapter, we review the execution model and architectural features of graphics processing units. Graphics processing units (GPUs) are highly parallel coprocessors, which specialize in data-parallel tasks. Data-parallel tasks can be subdivided into smaller tasks, which can be processed in parallel. For example, a blur filter in image processing can be applied on all pixels of the image simultaneously. Processors, which can execute a single instruction on multiple data items in parallel are well-suited for solving data-parallel tasks. In Flynn's taxonomy of parallel processors, this type of execution model is called single-instruction multiple data (SIMD) [3]. When attempting to write efficient programs, it is often beneficial to consider the execution model of GPUs to be SIMD. However, the actual execution model of GPUs is close to, but not purely SIMD [28], which we discuss in detail in Section 2.1.

Graphics processing units have traditionally been used for rendering real-time graphics [25, 27], where a scene is projected on the framebuffer by passing the input through various stages of a graphics pipeline as shown in Figure 2.1. The most computation-heavy stages of the graphics pipeline can be solved in a data-parallel fashion, which has been the driving force of the architecture of GPUs. During the vertex shader stage for example, the positions of vertices depicting a three-dimensional scene are projected into two-dimensional screen coordinates, or during the fragment shader stage, the colors of pixels are determined by sampling textures and using linear interpolation to compute the final color [27, 31, 32]. The tasks performed during both of these stages are data parallel, as the vertices or pixels can be processed independently.

Graphics processing units have been designed around a model called stream programming, where small GPU programs, *kernels*, operate on streams of data, where each stream element can be processed independently [61]. In contrast to central

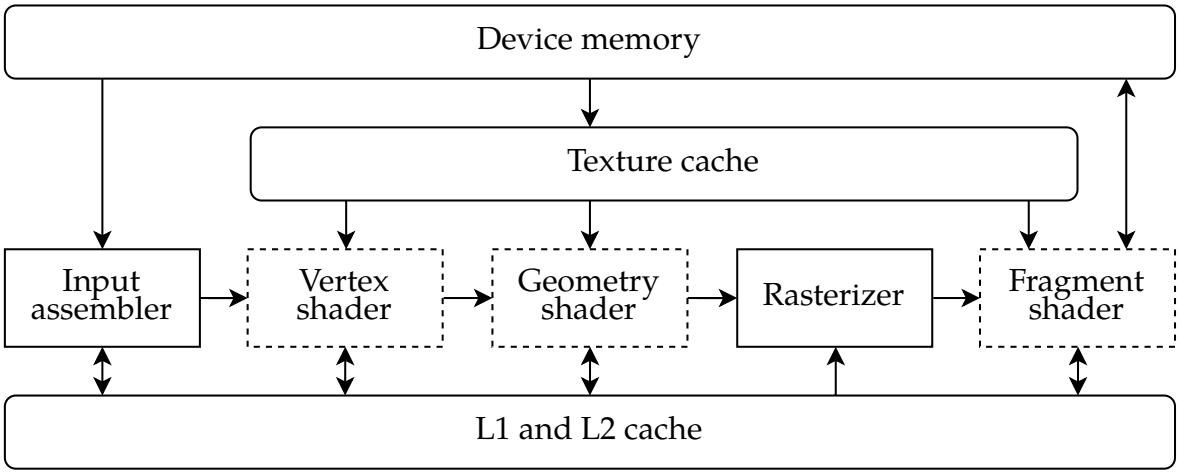


Figure 2.1: A simplified graphics pipeline adapted from [9, 27, 32]. Programmable stages are marked with a dashed line. During input assembly the data structures required at later stages are gathered from device memory and passed to the vertex shader [27]. If possible, intermediate data is cached to L1 and L2 between stages. Vertex, geometry and fragment shader stages are executed on a grid of generic stream processors [25, 60]. Compute shaders used for general-purpose computing are analogous to the other programmable stages in the pipeline and executed on the same processors, with the difference that compute pipelines consist of only one stage [32]. We focus on compute shaders in this work and refer the reader to [27] for a more detailed explanation of the stages in a graphics pipeline.

processing units which have to excel in general tasks, the architecture of GPUs has been designed to maximize the throughput of the operational and memory systems in tasks expressed with the stream programming model [25]. As this programming model exposes the parallelism and data dependencies of a program, computational throughput can be increased by employing a large number of parallel stream processors [61] and wide memory buses. However, the cost of utilizing wider memory buses increases memory access latency [25]. To mitigate the penalty of longer access latency, hardware multithreading is used to improve instruction-level and thread-level parallelism during execution, in addition to employing multilevel caches to reduce traffic to off-chip memory [61, 62].

Operational and memory access latencies are hidden, when an instruction can be issued at every clock cycle as shown in Figure 2.2, or the memory systems are fully utilized. While in the introduction we discussed the disproportional increase in compute performance with respect to the memory systems as noted by Wulf [23], in 2004, Patterson [63] made the remark that latency improves with a slower rate than memory bandwidth. Because throughput has been prioritized over latencies in GPU design, hiding latencies becomes a major consideration for both architecture

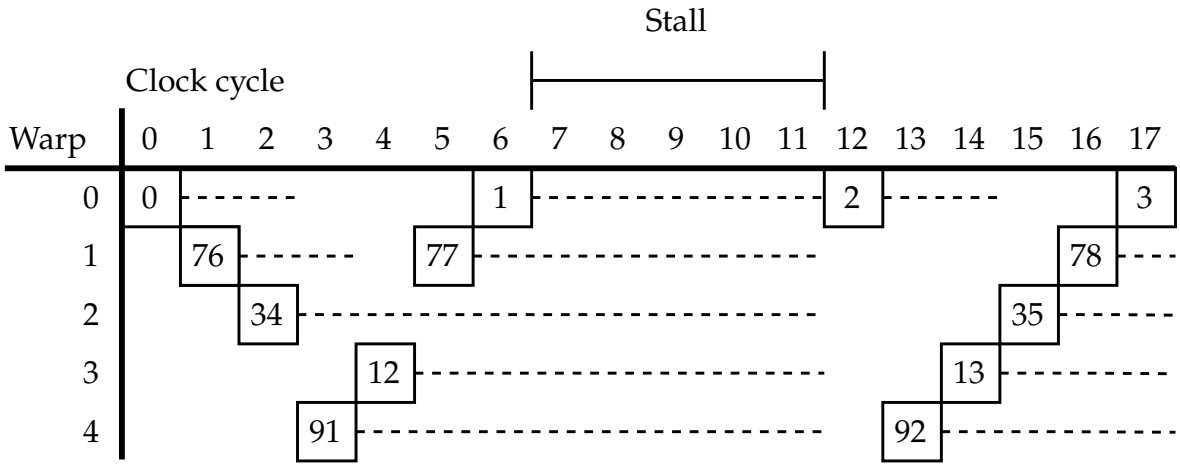


Figure 2.2: An example of instruction pipelining with hardware multithreading. At each clock cycle, an instruction is fetched and dispatched to the functional units if possible. The number inside a square indicates the number of the instruction dispatched. The time that it takes for an instruction to complete is marked with a dashed line. The pipeline is stalled if none of the threads is ready to run. Instruction execution latency can be hidden either by multithreading a large number of threads [9, 25], or ensuring, that there are sufficiently many subsequent instructions that can be executed independently [64, 65].

designers and programmers developing software for GPUs.

In this work, we focus on how GPUs can be used to solve stencil operations on workloads commonly found in scientific computations. Traditionally, GPUs have been geared towards sampling two-dimensional textures with low-order stencils, where data have been stored as single-precision floating-point numbers packed into four elements [25]. For this purpose, linear interpolation used with sampling is generally implemented in hardware [66]. More complicated stencils with different coefficients must be implemented in software.

As high-performance computing on GPUs gained popularity, both AMD and NVIDIA introduced a line of GPUs tuned for scientific computations[1]. Notably, these GPUs feature larger caches and more double-precision arithmetic units than consumer-grade GPUs. There are typically twice as many single-precision arithmetic logic units than double-precision arithmetic logic units [67], which is in balance with the increased memory traffic required for supplying 8-byte data words for arithmetic operations. For the rest of this work, we focus on discrete NVIDIA GPUs tuned for scientific computations, which are based on the Pascal microarchitecture.

Modern GPUs are stated to conform fully to the IEEE 754-2008 standard [68] for binary floating-point arithmetic [69]. However, differing from x86 CPU architectures,

[1] AMD FireStream and NVIDIA Tesla.

Table 2.1: Common terms used in GPU computing.

CUDA [9]	**OpenCL** [10]	**This work**
Thread	Work-item	Thread
Warp	Wavefront	Warp
Thread block	Work-group	Cooperative thread array
Streaming multiprocessor	Compute unit	Multithreaded SIMT processor
CUDA Core	Processing element	Stream processor

floating-point exceptions are not handled with GPU hardware [69]. Additionally, at compile time, the rounding mode and the accuracy of arithmetic operations used in GPU kernels can be changed, which may be used to trade accuracy for performance [66]. Floating-point arithmetic on GPUs is discussed more in detail in Section 5.1.

2.1 GPU architecture

In this section, we give an overview of GPU architectures from the perspective of general-purpose computing. Hardware details unique to graphics processing are left out for simplicity. While our focus is on the architecture of NVIDIA GPUs, AMD GPUs rely on the same general architecture and techniques to hide latencies as discussed in the previous section [70]. We refer the reader to Table 2.1 for a translation of the terms associated with CUDA, OpenCL and the terms used in this work.

Let us review the necessary concepts before discussing GPU architectures in detail. A thread is a lightweight process which has its own program counter and memory space [2]. Generally only one thread executes at a time on a processor. With multithreading, a thread is assigned a time slot to execute instructions on the processor before being switched out to another. We refer to this type of execution as concurrency. The action of switching the active thread to another is called a context switch. In a context switch, the execution context of the active thread is stored in a stack and the execution of another thread is resumed. A stack is an area in high-speed memory which serves as a temporary storage for local variables and other data [2, 3] and the execution context contains the resources specific to some thread, which generally include the local variables and the program counter. Multithreading can be coarse-grained, where several instructions of a thread are executed before a

context switch, or fine-grained, where instructions from different threads are issued at every clock cycle [3].

CUDA threads have some key differences when compared with traditional threads executed on a CPU. While CUDA threads have their own program counter and memory space, and they execute scalar instructions, GPU hardware can issue only one to two instructions per clock cycle, which are broadcast to a group of threads and executed in parallel [25]. This group is called a *warp*, and it consists of usually 32 threads. Graphics processing units execute warps in a fine-grained fashion, where instructions are fetched and issued for different warps at every clock cycle [25, 62]. A warp is analogous to a thread of SIMD instructions [3] with the exception that the if-then-else construct is supported at thread-level granularity, which allows the threads of a warp to follow different execution paths [25]. However, if the threads of a warp follow different execution paths, then the instructions for the diverging threads must be issued serially. This is done by masking threads not taking part in the instruction as inactive, while active threads execute the broadcast instruction [62]. This type of execution model has been called single instruction, multiple threads (SIMT) to draw a distinction of a purely SIMD-style execution, where a vector instruction is executed on a vector of data [3, 25]. However, in order to fully utilize the parallel processors of the hardware, instructions within a warp have to follow the same execution path [9, 25, 62, 66].

Graphics processing units accommodate several multithreaded SIMT processors, which execute warps concurrently (Figures 2.3 and 2.4). A block of warps, where warps can communicate via on-chip caches is called a cooperative thread array (CTA) [25]. At the launch of a kernel, a grid of CTAs is distributed to the multithreaded SIMT processors of the GPU, where the CTAs are executed independently on each SIMT processor.

There are several types of memories dedicated to each SIMT processor. First, there is a high-speed data cache, called the unified cache, which functions as a L1 and texture cache [8]. Loads from device memory are generally cached implicitly in the unified cache [67]. The L1 portion of the cache is also used for storing the local variables of threads [67]. In contrast to multilevel caches on a CPU, which are large enough to hold entire working sets in memory, GPU caches are primarily used for reusing data local to a CTA in order to reduce traffic to off-chip memory [3]. The texture cache is read only, and optimized for memory reads exhibiting two-dimensional spatial locality [9]. Memory accesses are spatially local if neighboring memory addresses are likely to be called with subsequent instructions.

Cache replacement policies are used to determine which cache line to discard at

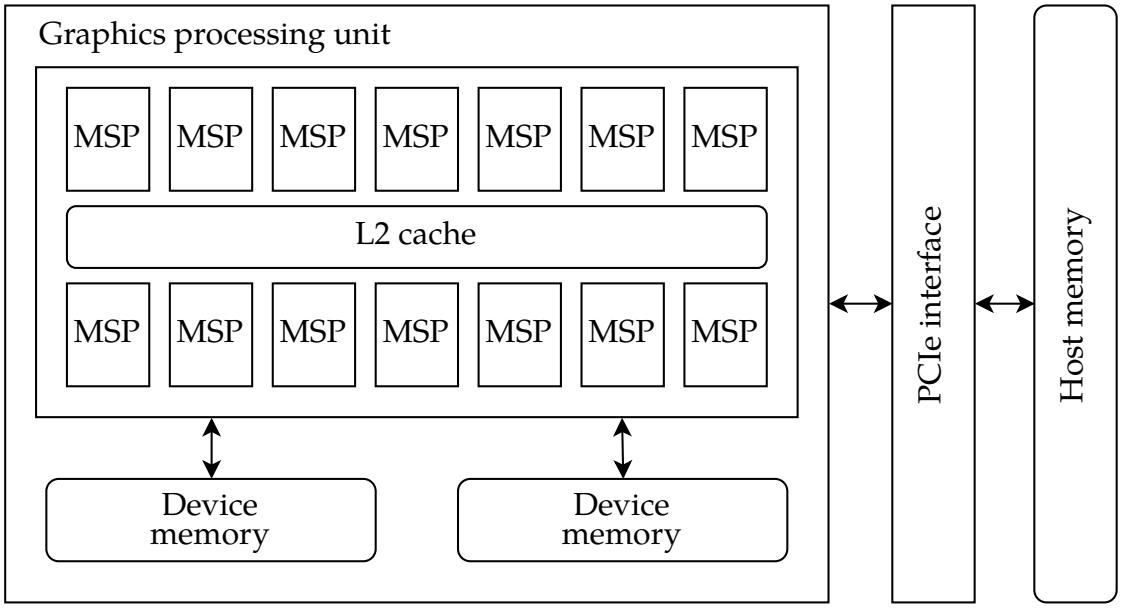


Figure 2.3: An illustration of GPU hardware architecture. A Pascal P100 PCIe GPU houses 56 multithreaded SIMT processors (MSP), which share a 4096 KiB L2 cache and a total of 16 GiB of device memory [8]. The device memory consists of four HBM2 memory packages and eight memory controllers, which supply a total of 4096-bit wide memory bus and the theoretical maximum memory bandwidth of 682 GiB/s [26].

a cache miss. Least-recently used (LRU) is a caching policy, where the least-recently used cache line is discarded when a cache miss occurs. A cache miss can happen for three reasons. If the cache is empty, a compulsory cache miss is encountered [3]. A capacity miss is a miss, where the cache is full and the queried data is not found in the caches [3]. Finally, a conflict miss happens when there is a collision in mapping data from two different addresses to the same cache line [3]. With n-way set-associative caches, cache lines are grouped into sets, where each set contains n cache lines [2]. The hit rate of the cache can be improved by increasing the number of cache lines per set, as data can be placed on any of the n cache lines in that set, which in turn reduces conflict misses [2]. However, caches with higher associativity require more complex and expensive hardware [2].

Previous work suggests, that the caching policy used with the Pascal microarchitecture is LRU, the cache is set-associative with four sets, the size of the L1 cache line is 32 bytes and cache lines are replaced in segments of 128 bytes [26, 67, 74]. Threads of a CTA also have access to a user-programmable data cache called shared memory, which is partitioned to a separate memory space [67]. Previous work suggests, that access latency to shared memory is 24 clock cycles if there are no conflict misses,

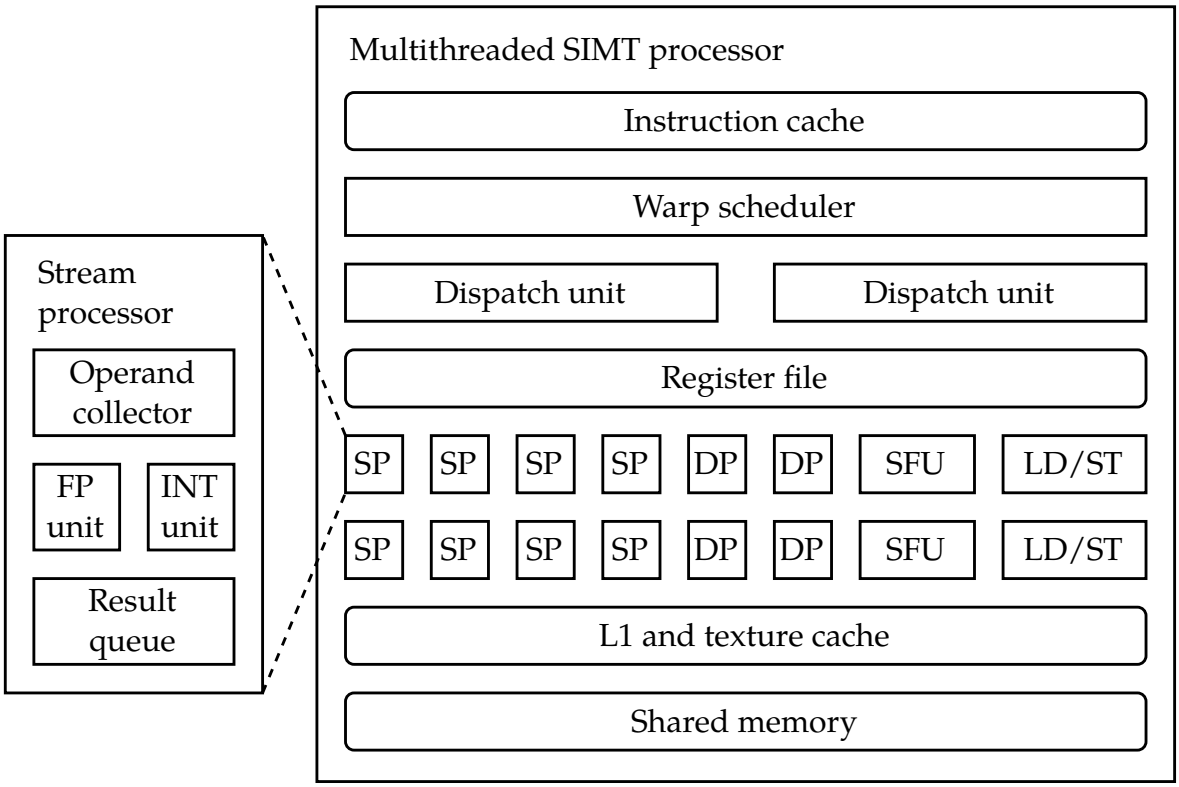


Figure 2.4: An illustration of a multithreaded SIMT processor. A Pascal P100 PCIe GPU houses 64 single-precision (SP) and 32 double-precision stream processors (DP), eight special function (SFU) and load-store units (LD/ST), a single warp scheduler and two dispatch units [8]. The register file can hold 128 KiB of data and shared memory 64 KiB. In previous work, the sizes of the L1 instruction and data caches have been suggested to be 8 KiB and 24 KiB, respectively [26]. Illustration of the stream processor has been adapted from the documentation for older Fermi architectures [71], however, we could not find any indication that the composition of the stream processor has been changed on Pascal architectures. NVIDIA explicitly states, that a stream processor on Pascal contains both the 32-bit floating-point and integer units, unlike on newer Volta and Turing architectures, where the units have been separated to dedicated stream processors [72, 73].

whereas hit latency to the L1 data cache is 82 cycles [26].

There is also a register file and an instruction cache on each multithreaded SIMT processor. The execution states of the threads being multithreaded on a GPU are held in on-chip caches and registers for the lifetime of the threads, therefore context switching is stated not to incur a performance penalty [9]. Because the execution states of threads are held in caches, the number of threads that can be run on a multithreaded SIMT processor at a time is limited by the size of the caches. Once the threads in a cooperative thread array terminate and the resources become available, new CTAs are launched on the SIMT processor [9].

The multithreaded SIMT processors of a GPU share an L2 cache and main memory. We refer to the main memory of the GPU as *device memory* throughout this work to make a distinction between the main memory of the host CPU. Device memory is aligned to segments of 128 bytes, and is accessed with 32-, 64-, and 128-byte transactions [9]. Modern GPUs utilize high-bandwidth memory (HBM), where the memory chips are stacked in a three-dimensional integrated circuit [67]. Accesses to device memory are implicitly cached in the L1 and L2 [9]. L2 is used also for storing instructions and constants in addition to data [26].

In addition to memories, a multithreaded SIMT processor houses a large number of stream processors, special function units, and load-store units as shown in Figure 2.4. A stream processor houses a single-precision floating-point and integer arithmetic-logic units [9, 72]. As single-precision and integer arithmetic units are located on the same stream processor, single-precision and integer instructions cannot be executed during the same cycle on Pascal architectures. There are also stream processors which execute double-precision floating-point instructions. Stream processors are not capable of any complex logic, such as branch prediction unlike cores of a CPU [25]. Special function units execute hardware-accelerated approximations of otherwise expensive instructions, such as the reciprocal square root and trigonometric functions [25]. Load-store units execute instructions used to read and write data to and from memories.

A multithreaded instruction fetch and issue unit consists of a warp scheduler and two dispatch units. At each cycle, a ready-to-run warp is selected from the warps of a CTA. For example, if there are 128 stream processors on a multithreaded SIMT processor, then there must be 4 warp schedulers and at least 128 threads to saturate the stream processors with work during a clock cycle. As there are two dispatch units per a multithreaded fetch and issue unit, two instructions can be broadcast to the execution units of the SIMT processor, as long as those instructions are executed on separate execution units [9]. For example, an arithmetic and load-store instruction

of a single thread can be dispatched during the same cycle.

Discrete GPUs are generally connected to the host system via a PCIe bus, which is used to transfer data between host and device memories, and to queue kernels to be executed on the GPU. PCIe bus may also be used for communication among GPUs in the node. The PCIe 3.0 x16 bus of a Tesla P100 PCIe GPU provides a directional bandwidth of 15 GiB/s [75]. There are also special interconnects, NVIDIA NVLink and AMD Crossfire, which can be used to connect 2–4 GPUs in a node. NVIDIA NVLink is stated to provide 19 GiB/s bandwidth per direction between pairs of Tesla P100 PCIe GPUs [8, 26, 75].

2.2 Programming GPUs

Graphics processing units have traditionally been programmed using application-programming interfaces (APIs) specialized in computer graphics, such as OpenGL [76] and Direct3D [27]. While modern graphics APIs provide compute shaders for solving general tasks, APIs created for general-purpose computing, such as CUDA [9] and OpenCL [10], offer more convenient abstractions of the GPU hardware for scientific computations with high-performance computers. In this section, we review the compilation stages from CUDA to assembly-level code, give an introduction to programming GPUs with CUDA, and discuss common caching techniques used to reduce traffic to off-chip memory.

Graphics processing units are independent coprocessors which require commands from a host CPU to function. During the execution of a program, the host queues commands via the PCIe bus or network to the GPU device, which are then executed asynchronously on the GPU [32]. CUDA runtime is a high-level API and a programming model, which extends the syntax of C++ with functions that can be used to control NVIDIA GPUs [77]. It is built on top of the CUDA driver API, which offers more explicit control over the device [9]. The abstraction levels of the CUDA driver and OpenCL APIs are similar, because with both APIs, the device context must be initialized, and GPU kernels loaded explicitly [10, 78]. With the CUDA runtime API, these actions are performed implicitly, which results in less verbose code [9]. In further discussion on CUDA, we refer to the CUDA runtime API unless otherwise mentioned.

A CUDA program contains code for both the host and device, which is compiled into an executable binary with the NVIDIA CUDA compiler. During intermediate compilation stages, the device-only code is compiled to either Parallel thread execu-

tion (PTX) code or a CUDA binary (cubin) [77]. Parallel thread execution is a virtual instruction set architecture (ISA), which provides an intermediate assembly language that can be further compiled to device-specific assembly just in time by the CUDA driver [9, 79]. CUDA binaries can be disassembled to streaming assembly (SASS) code, which contains instructions native to some specific GPU ISA (Listing 2.3). The host code written in C++ is compiled into an object file, where the generated PTX and cubin files are embedded [77].

```
#define VALUE_COUNT (256)

int
main(void)
{
    float values[VALUE_COUNT];

    for (int index = 0; index < VALUE_COUNT; ++index)
        values[index] = index * index;

    return 0;
}
```

Listing 2.1: An example of a loop, where an operation is applied to all elements in an array on a CPU.

```
#define VALUE_COUNT (256)

__global__ void
kernel(float* values)
{
    const int index = threadIdx.x + blockIdx.x * blockDim.x;

    if (index < VALUE_COUNT)
        values[index] = index * index;

}

int
main(void)
{
    float* values;
    cudaMallocManaged(&values, VALUE_COUNT * sizeof(values[0]));

    const int threads_per_cta = 128;
    const int ctas_per_grid = ceil((float) VALUE_COUNT / threads_per_cta);

    kernel<<<ctas_per_grid, threads_per_cta>>>(values);
    cudaDeviceSynchronize();

    cudaFree(values);
    return 0;
}
```

Listing 2.2: An example of a CUDA program, which is logically equivalent to the program shown in Listing 2.1.

```
code for sm_60
  Function : _Z6kernelPf
.headerflags    @"EF_CUDA_SM60 EF_CUDA_PTX_SM(EF_CUDA_SM60)"

        /*0008*/                   MOV R1, c[0x0][0x20] ;
        /*0010*/                   S2R R0, SR_TID.X ;
        /*0018*/                   S2R R2, SR_CTAID.X ;
        /*0028*/                   XMAD R0, R2.reuse, c[0x0] [0x8], R0 ;
        /*0030*/                   XMAD.MRG R3, R2.reuse, c[0x0] [0x8].H1, RZ ;
        /*0038*/                   XMAD.PSL.CBCC R2, R2.H1, R3.H1, R0 ;
        /*0048*/                   ISETP.GT.AND P0, PT, R2, 0xff, PT ;
        /*0058*/                @P0 EXIT ;
        /*0068*/                   XMAD R0, R2, R2, RZ ;
        /*0070*/                   XMAD.MRG R3, R2, R2.H1, RZ ;
        /*0078*/                   XMAD.PSL.CBCC R0, R2.H1, R3.H1, R0 ;
        /*0088*/                   I2F.F32.S32 R0, R0 ;
        /*0090*/                   SHR R4, R2.reuse, 0x1e ;
        /*0098*/                   ISCADD R2.CC, R2, c[0x0][0x140], 0x2 ;
        /*00a8*/                   IADD.X R3, R4, c[0x0][0x144] ;
        /*00b0*/                   STG.E [R2], R0 ;
        /*00d0*/                   EXIT ;
        /*00d8*/                   BRA 0xd8 ;
```

Listing 2.3: The SASS disassembly of the CUDA kernel shown in Listing 2.2. We refer the reader to [80] for a description of the Pascal instruction set. No-operation (NOP) instructions have been removed for clarity.

Next, we introduce the basics of programming in CUDA. A cooperative thread array (CTA) is a collection of warps, which are multithreaded on a single multithreaded SIMT processor, and which share the resources available on that processor, such as the L1 cache and registers. On NVIDIA devices, there are 32 threads per warp, where a thread corresponds to the portion of work executed on one of the stream processors or other executional units. Each thread may be assigned a maximum of 255 registers, and a thread can access registers of other threads of the same warp with a shuffle instruction [9].

At the launch of a kernel, the caller defines the number of threads per CTA, and the total number of CTAs to be executed on the device. Each thread is assigned a unique identifier, analogous to the value of a counter in a `for` loop. Examples of logically equivalent programs in C and CUDA are shown in Listings 2.1 and 2.2.

Finally, we introduce some common caching techniques for GPUs and discuss the related issues. The processing power of GPUs in terms of floating-point operations per second is significantly higher than what can be served with the memory system. With a Tesla P100 PCIe GPU for example, which runs at 1328 MHz clock frequency, contains 56 active SIMT processors, which each can issue 128 single-precision floating-point instructions per cycle [8, 26, 75], the peak arithmetic performance is $W = 1328 \cdot 10^6 \cdot 56 \cdot 128 = 9.5 \cdot 10^{12}$ floating-point operations (flops) per second. Given double-data rate HBM2 memory running at 715 MHz supplying data via a

4096-bit wide memory bus [8, 26, 81], the theoretical maximum memory bandwidth is $Q = 715 \cdot 10^6 \cdot 4096 \cdot 2/8 = 682\,\text{GiB/s}$. Therefore the operational intensity of a kernel has to be at least $W/Q = 13$ floating-point operations per each byte transferred to reach the peak arithmetic performance. In many cases, the operational intensity of a kernel is lower than what is needed to saturate the floating-point units, in which case the kernel becomes inherently bound by memory bandwidth. If there are enough instructions in flight to hide latencies in such kernel, techniques that reduce the need to fetch data from device memory are necessary to improve performance further.

Cache blocking is a technique, where a problem is solved in small subsets, such that the working set at any given time is small enough to fit in caches [3]. An example of a cache blocking techniques used with GPUs is tiled rendering [82], where an image is constructed by rasterizing the scene as tiles where the working set, such as triangle data and a portion of a texture, is small enough to fit into the on-chip caches. However, generally it is not recommended to rely on implicit caching as multiple CTAs compete for the same resources. If the memory fetches of CTAs span too many addresses, this would lead to thrashing, where capacity and conflict misses lead to cache lines being continuously replaced.

For this purpose, GPUs house an on-chip cache, called shared memory, which can be explicitly managed by the programmer [9]. Shared memory can be used to communicate data among threads of a CTA. However, the threads in a CTA have to be synchronized before accessing the data in shared memory in order to avoid a data race. Synchronization in turn incurs an overhead. Length of the overhead varies case-by-case and depends on the execution state of the warps on the SIMT processor before synchronization. The exact details on how thread scheduling is implemented on the hardware is not publicly available, which makes it difficult to make any further assumptions on the overhead. The amount of shared memory assigned to a CTA limits the number of threads that can be executed on a SIMT processor at a time. This can be problematic in cases, where a large amount of data needs to be held in caches in order to benefit from reuse. Assigning a large amount of shared memory for a CTA reduces the number of threads that can be executed concurrently on a multithreaded SIMT processor, which in turn makes the processor more susceptible to stalls.

Chapter 3

Magnetohydrodynamics

In this chapter, we review the equations and methods used in this work to simulate magnetohydrodynamics. Instead of testing our library and domain-specific language in idealized test cases, we use magnetohydrodynamics to demonstrate that the tools developed for this work are effective in solving complex problems encountered in computational physics and high-performance computing. Next, we give an overview of magnetohydrodynamics and detail the equations solved during the simulation. In Section 3.1, we describe how the simulation domain is discretized on a structured grid, and how derivatives are solved with the finite-difference method. We also discuss the memory access patterns arising from this differentiation method, which are an integral part of the discussion related to performance optimizations later in this work. Finally, in Section 3.2, we describe the integration scheme used to advance the simulation in time.

Magnetohydrodynamics (MHD) is the study of electrically conducting fluids and plasmas which interact with a magnetic field [83]. MHD simulations have a wide range of applications, especially in astrophysics, where simulations of electrically conducting liquids and plasmas are a major tool for understanding the evolution of stars and galaxies [11]. In such simulations, several fields have to be updated in order to advance the state of the fluid in time. In our case, there are a total of eight fields; the components of three-dimensional velocity $\mathbf{u}$ and magnetic vector potential $\mathbf{A}$, in addition to specific entropy s and density ρ, which are scalar fields. If a field interacts with another, that is, updating one field requires knowledge of another, we say that those fields are coupled. Characteristic of magnetohydrodynamics, there is a strong two-way coupling between the magnetic field $\mathbf{B}$ and velocity of the fluid $\mathbf{u}$, where both fields drive one another [83].

Next, we show how the rate of change is solved for each field when simulating

the flow of compressible fluids with an ideal equation of state. We closely follow the equations used in the Pencil Code [19]. Let D/Dt be the convective derivative $D/Dt = \partial/\partial t + (\mathbf{u} \cdot \nabla)$ and $\ln \rho$ the logarithmic density. The Laplace operator is denoted as ∇^2 and the curl operator as $\nabla\times$. We can now write the rate of change for $\ln \rho$ with the continuity equation

$$\frac{D \ln \rho}{Dt} = -\nabla \cdot \mathbf{u} \, . \tag{3.1}$$

Next, we construct the rate of change for the magnetic vector potential $\mathbf{A}$. First, the magnetic flux density $\mathbf{B}$ is given by

$$\mathbf{B} = \nabla \times \mathbf{A} \, . \tag{3.2}$$

Furthermore, let η be the magnetic diffusivity and μ_0 the magnetic vacuum permeability. The electric current density $\mathbf{j}$ is then given by

$$\begin{aligned} \mathbf{j} &= \nabla \times \mathbf{B}/\mu_0 \\ &= \nabla \times (\nabla \times \mathbf{A})/\mu_0 \\ &= [\nabla(\nabla \cdot \mathbf{A}) - \nabla^2 \mathbf{A}]/\mu_0 \, . \end{aligned} \tag{3.3}$$

Finally, we can use Faraday's law together with Equations 3.2 and 3.3 to compute the rate of change of $\mathbf{A}$

$$\frac{\partial \mathbf{A}}{\partial t} = \mathbf{u} \times \mathbf{B} - \eta \mu_0 \mathbf{j} \, . \tag{3.4}$$

As the gradient of an arbitrary scalar field can be added to the magnetic vector potential, various reformulations of this equation exist. In this work, we have used the Weyl gauge, where the electric scalar potential is $\Phi = 0$ [19].

Next, we define the equations for updating velocity $\mathbf{u}$. Let $\mathbf{S}$ be the traceless rate-of-shear tensor

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3}\delta_{ij} \nabla \cdot \mathbf{u}\right) . \tag{3.5}$$

Furthermore, heat capacity at constant pressure c_p, heat capacity at constant volume c_v, kinematic viscosity ν, and bulk viscosity ζ be some scalar constants. The speed of sound c_s for perfect, non-isothermal gases is defined as

$$c_s^2 = c_{s0}^2 \exp\left[\frac{\gamma s}{c_p} + (\gamma - 1) \ln \frac{\rho}{\rho_0}\right] , \tag{3.6}$$

where γ is the adiabatic index $\gamma = c_p/c_v$ and s the specific entropy defined later in Equation 3.12 [19]. We can now write the Navier-Stokes equation for compressible, non-isothermal flow as

$$\begin{aligned}\frac{D\mathbf{u}}{Dt} = &- c_s^2 \nabla\left(\frac{s}{c_p} + \ln\rho\right) + \frac{\mathbf{j}\times\mathbf{B}}{\rho} \\ &+ \nu\left[\nabla^2\mathbf{u} + \frac{1}{3}\nabla(\nabla\cdot\mathbf{u}) + 2\mathbf{S}\cdot\nabla\ln\rho\right] + \zeta\nabla(\nabla\cdot\mathbf{u})\,.\end{aligned} \tag{3.7}$$

For computing the rate of change for entropy s, we first have to define the contract operator $\otimes$ as

$$\mathbf{S}\otimes\mathbf{S} = \sum_{i=1}^{3}\sum_{j=1}^{3} S_{ij}^2\,. \tag{3.8}$$

Let $\mathcal{H}$ and C be explicit heating and cooling terms, respectively, which may depend on ρ or temperature T. In this work, we assume that $\mathcal{H}$ and C are constant. Finally, the radiative thermal conductivity is given by K and the thermal diffusivity by $\chi = K/(\rho c_p)$ [19]. For the heat conduction term $\nabla\cdot(K\nabla T)/\rho T$ holds that [19]

$$\begin{aligned}\frac{\nabla\cdot(K\nabla T)}{\rho T} &= c_p\chi[\gamma\nabla^2 s/c_p + (\gamma-1)\nabla^2\ln\rho] \\ &+ c_p\chi[\gamma\nabla s/c_p + (\gamma-1)\nabla\ln\rho] \\ &\cdot[\gamma(\nabla s/c_p + \nabla\ln\rho) + \nabla\ln\chi]\,.\end{aligned} \tag{3.9}$$

In this work, we assume a constant K and compute

$$\nabla\ln\chi = -\nabla\ln\rho\,. \tag{3.10}$$

Additionally, we calculate $\ln T$ as

$$\ln T = \ln T_0 + \frac{\gamma s}{c_p} + (\gamma-1)(\ln\rho - \ln\rho_0)\,, \tag{3.11}$$

where $\ln T_0$ and $\ln\rho_0$ are some constants. Finally, we can write the rate of change for specific entropy s as [19]

$$\rho T\frac{Ds}{Dt} = \mathcal{H} - C + \nabla\cdot(K\nabla T) + \eta\mu_0\mathbf{j}^2 + 2\rho\nu\mathbf{S}\otimes\mathbf{S} + \zeta\rho(\nabla\cdot\mathbf{u})^2\,. \tag{3.12}$$

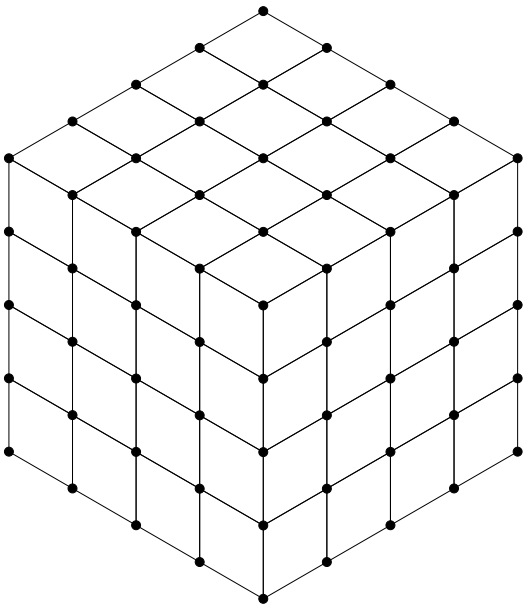


Figure 3.1: An equidistantly spaced structured grid.

3.1 Finite-difference methods

In this section, we show how the equations presented in the previous section can be solved with the finite-difference method. With this method, continuous fields are mapped onto a discrete grid, where the derivatives at the position of each vertex can be calculated by sampling neighboring vertices. In this work, we focus on uniformly spaced, structured grids, where the distance between the vertices is constant and each vertex, except the ones at the boundary, is connected to the same number of neighbors. The type of grid used in this work is visualized in Figure 3.1.

The set of points that are used to approximate solutions to partial differential equations depends on the form of the difference equation. A difference equation can be written as forward, central and backward differences, where only the points in front of the target point, on both sides, or behind the target point are sampled and weighted with some finite-difference coefficients, respectively [14]. Recall the definition of derivatives

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h} , \tag{3.13}$$

where f is some continuous real-valued function, and h is the spacing between samples. In practical applications, $f'(x_0)$ cannot be solved exactly, which introduces an error to the calculation because h has to be some finite value. A finite-difference equation is said to be of order n, when the discretization error has order $O(h^n)$.

Here we use the big O notation to capture the error term with the greatest power of h [14]. Next, let us derive the error for first-order forward differences from Taylor's theorem [14].

$$\begin{aligned} f(x_0 + h) &= \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)h^n}{n!} \\ &= f(x_0) + f'(x_0)h + \sum_{n=2}^{\infty} \frac{f^{(n)}(x_0)h^n}{n!} \\ &= f(x_0) + f'(x_0)h + O(h^2) \,. \end{aligned} \tag{3.14}$$

By moving $f'(x_0)$ to the left-hand side, we get

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + O(h) \,. \tag{3.15}$$

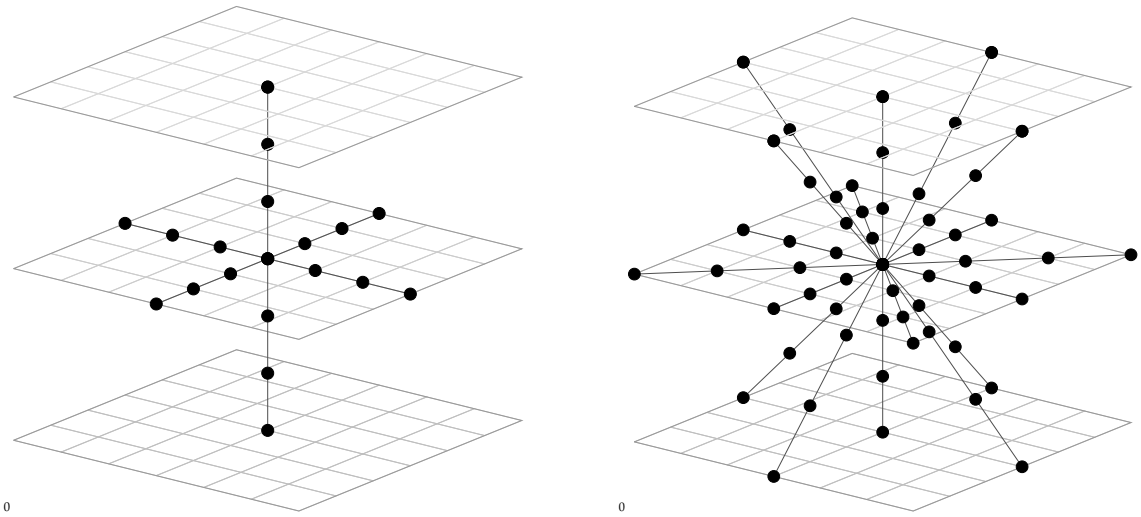
Here $O(h)$ is the truncation error, which is the error caused by representing an infinite series as a finite sum.

We can now express Equation 3.15 in a notation more suitable for representing computations in a discrete grid. Let f_i be the ith vertex of the continuous field f and δ_x the grid spacing with respect to axis x. The alternative representation of the equation for forward differences can now be written as

$$\frac{\partial f_i}{\partial x} = \frac{f_{i+1} - f_i}{\delta_x} + O(\delta_x) \,. \tag{3.16}$$

In this work, we use explicit high-order central differences, as this gives almost as high accuracy as more computationally intensive spectral methods [11]. Other methods than finite differences could also be used, such as the finite element or finite volume methods, but these are more difficult to implement and do not provide an unequivocal advantage over finite differences in cases where the simulation domain can be represented by a relatively simple, structured grid [11, 84]. Without loss of generality, the sixth-order central difference equations for first, second and mixed derivatives can be written for any axis as

$$\frac{\partial f_i}{\partial x} = \frac{(f_{i+3} - f_{i-3}) - 9(f_{i+2} - f_{i-2}) + 45(f_{i+1} - f_{i-1})}{60\delta_x} + O(\delta_x^6) \,, \tag{3.17}$$



(a) 19-point stencil (b) 55-point stencil

Figure 3.2: Visualization of a 19-point (a) and a 55-point (b) stencil.

$$\frac{\partial^2 f_i}{\partial x^2} = \frac{2(f_{i+3} + f_{i-3}) - 27(f_{i+2} + f_{i-2}) + 270(f_{i+1} + f_{i-1}) - 490 f_i}{180\delta_x} + O(\delta_x^6)\,, \tag{3.18}$$

and mixed derivatives may be calculated using a compact form [19]

$$\begin{aligned}\frac{\partial^2 f_{i,j}}{\partial x \partial y} = \frac{1}{720\delta_x\delta_y}\Big[&270(f_{i+1,j+1} - f_{i-1,j+1} + f_{i-1,j-1} - f_{i+1,j-1}) \\ &- 27(f_{i+2,j+2} - f_{i-2,j+2} + f_{i-2,j-2} - f_{i+2,j-2}) \\ &+ 2(f_{i+3,j+3} - f_{i-3,j+3} + f_{i-3,j-3} - f_{i+3,j-3})\Big] \\ &+ O(\delta_x^6, \delta_y^6)\,.\end{aligned} \tag{3.19}$$

As we can see, information about neighboring vertices has to be known in order to compute any of the derivatives for the vertex in the middle. This access pattern is called an n-point stencil, where n is the number of vertices whose values need to be known. A 19-point stencil required to solve Equations 3.17 and 3.18 is shown in Figure 3.2(a). A much larger stencil consisting of 55 points is required to also compute cross partial derivatives. The stencil required for solving Equations 3.17, 3.18 and 3.19 is shown in Figure 3.2(b).

Next, we define some additional terminology. The set of vertices updated during each pass over the grid is called the *computational domain*. Because the stencils of

vertices at the boundaries extend beyond the grid, the computation at boundaries requires special care. A common approach is to pad the boundaries with additional vertices, which are updated with some values depending on the boundary conditions and used only for reading when solving the partial differential equations [13, 19]. This padding is called a *ghost zone*. There are several benefits of using ghost zones. First, the boundary conditions can be computed separately from the integration kernel. With parallelized solvers, the boundaries can therefore be communicated simultaneously with updating the part of the computational domain, where the vertices in the ghost zone are not read. Second, derivatives can be computed more efficiently during the update pass, as branching and memory fetches from far-away addresses are not needed when updating vertices near the boundaries. Third, updating all vertices requires the same amount of work, which results in better load balancing. Finally, computing boundary conditions outside of the integration kernel is more convenient for the programmer, as complex logic, such as communication between devices can be managed in a separate function. The drawbacks of using ghost zones are increased memory consumption and redundant memory transactions required to update the halo after each integration substep. We use periodic boundary conditions throughout this work.

3.2 Runge-Kutta integration

In this work, we use an explicit low-storage Runge-Kutta method based on work by Williamson [59] to advance the state of the system in time. With explicit integration methods, the state of the system is advanced in time based solely on the previous state. However, while explicit methods are generally easier to implement than implicit methods, explicit integration may lead to an unstable system if the simulation parameters are not selected carefully.

Let δt be the length of the time step $\delta t = t_1 - t_0$. The time step δt is usually selected using an adaptive timestepping scheme based on the Courant-Friedrichs-Lewy condition, which ensures that δt is small enough for the system not to become unstable when using explicit integration methods [11, 19]. In this work, we leave adaptive timestepping schemes out of scope and use a constant time step. In contrast to naïve Euler integration, where the state is advanced δt units forward in time by computing the rate of change only once, with Runge-Kutta methods the rate of change is computed at several points in time in the interval (t_0, t_1) and weighted to obtain more accurate approximation of the solution [14]. Throughout this work,

we use the term *substep* to refer to a single pass over the grid, where the state of a vertex is advanced one intermediate step forward. For example, with third-order integration, we perform three substeps, which form a full integration step.

The formula for computing an integration step using the low-storage scheme described by Williamson [59] is shown in Equations 3.20 and 3.21. Here s is the substep number $s = 1, 2, ..., n$ and $w^{(s)}$ denotes the value in temporary storage w at substep s. Additionally, the initial state is given by $f^{(0)}$ and we set $\alpha^{(1)} = w^{(0)} = 0$. The scheme is written as

$$w^{(s)} = \alpha^{(s)} w^{(s-1)} + \delta t \left(\frac{\partial f}{\partial t} \right)^{(s-1)} \tag{3.20}$$

and

$$f^{(s)} = f^{(s-1)} + \beta^{(s)} w^{(s)} \, . \tag{3.21}$$

With our solver, we use a modified formulation of this scheme, shown in Equation 3.25. While our scheme requires the same amount of storage as Williamson's original 2N scheme, the modified scheme requires less memory transactions, as a substep can be solved in a single pass over the vertices of the grid, eliminating the need to write intermediate values in w back to device memory. With the modified scheme, race conditions are avoided by reading from and writing to distinct arrays. While we expect that this modified scheme has been discovered before, it is not widely documented and we could not identify the scheme from previous work [85–87]. Proof of our formulation follows. Recall Williamson's integration scheme shown in Equations 3.20 and 3.21. We show that this scheme can be written in a form where $f^{(s)}$ can be solved in a way, where additional storage is not needed to hold the intermediate result $w^{(s)}$. Substituting Equation 3.20 for $w^{(s)}$ in Equation 3.21, we get

$$f^{(s)} = f^{(s-1)} + \beta^{(s)} \left(\alpha^{(s)} w^{(s-1)} + \frac{\partial}{\partial t} f^{(s-1)} \delta t \right) \, . \tag{3.22}$$

Furthermore, Equation 3.21 can be written in the form

$$w^{(s-1)} = \frac{f^{(s-1)} - f^{(s-2)}}{\beta^{(s-1)}} \, . \tag{3.23}$$

Substituting this, we can rewrite Equation 3.22 as

$$f^{(s)} = f^{(s-1)} + \beta^{(s)} \left(\alpha^{(s)} \frac{f^{(s-1)} - f^{(s-2)}}{\beta^{(s-1)}} + \frac{\partial}{\partial t} f^{(s-1)} \delta t \right) \, . \tag{3.24}$$

For the initial step it holds that $\alpha^{(1)} = 0$. This allows us to write the integration scheme in the form

$$f^{(s)} = \begin{cases} f^{(s-1)} + \beta^{(s)} \frac{\partial}{\partial t} f^{(s-1)} \delta t, & \text{if } s = 1 \\ f^{(s-1)} + \beta^{(s)} \left[\alpha^{(s)} \frac{f^{(s-1)} - f^{(s-2)}}{\beta^{(s-1)}} + \frac{\partial}{\partial t} f^{(s-1)} \delta t \right], & \text{otherwise, } \square. \end{cases} \tag{3.25}$$

This scheme requires only two arrays to be held in memory; one for storing the current state, and one for storing the previous or the next state. As no stencil operations are performed with the field $f^{(s-2)}$ in Equation 3.25, the array storing $f^{(s-2)}$ can be overwritten with the state $f^{(s)}$ after $f^{(s-2)}$ has been read.

In this chapter, we gave an overview of the equations and methods used to drive magnetohydrodynamics simulations. Magnetohydrodynamics serves as a good test case particularly because of the number of coupled fields. When computing the equations, finding an efficient way to cache portions of the fields is challenging and not explored adequately in previous work. Cache size limits the number of data items that can be reused during computation, and blocking techniques commonly used with simpler, axis-aligned stencils cannot be used because of the limited cache size and registers. In addition to updating coupled fields, we also have to compute at least first, second and cross partial derivatives, which are especially challenging to solve in a way, where as much of the data could be reused in caches when using high-order differentiation schemes. These requirements encompass many applications involving stencil computations, such as any other simulation where first- and second-order partial differential equations have to be solved to advance the state of the system.

Chapter 4

Library for stencil computations

The contributions of this work are a library for processing three-dimensional stencils, a domain-specific language for expressing stencil computations on a high-level, and a compiler for generating efficient CUDA kernels from source files written in this domain-specific language. In this chapter, we present these contributions and discuss our design decisions. In Section 4.1, we present the architecture of the library and its interface, and give examples how to use it for solving a timestep. In Section 4.2, we present the grammar of the domain-specific language and show examples of the syntax of programs written in the language. Finally, in Section 4.3, we discuss the technical details of our compiler implementation, review the compilation stages from translating source code written in the domain-specific language to kernels usable with our library and give a thorough description of the algorithm skeleton and optimizations performed during the generation of the stencil kernel.

Our primary motivation for creating the suite was to utilize the GPU hardware efficiently in three-dimensional stencil computations. We have focused especially on the demands in computational sciences, where large stencils are often-times used to achieve sufficient accuracy. As computations with large stencils are challenging to solve efficiently on hardware where the size of the cache is not large enough to hold the working set in on-chip memory, our focus has especially affected the way we generate kernels and which type of optimizations should be applied. As efficient programming of GPUs is also relatively verbose and requires expertise in the execution model and hardware features, we decided to create a domain-specific language which can be used to express a wide range of tasks in computational sciences. For the needs in high-performance computing, the library has been designed in a way which allows it to be extended to support multiple devices in future work.

The suite presented in this chapter is licenced under the GNU General Public

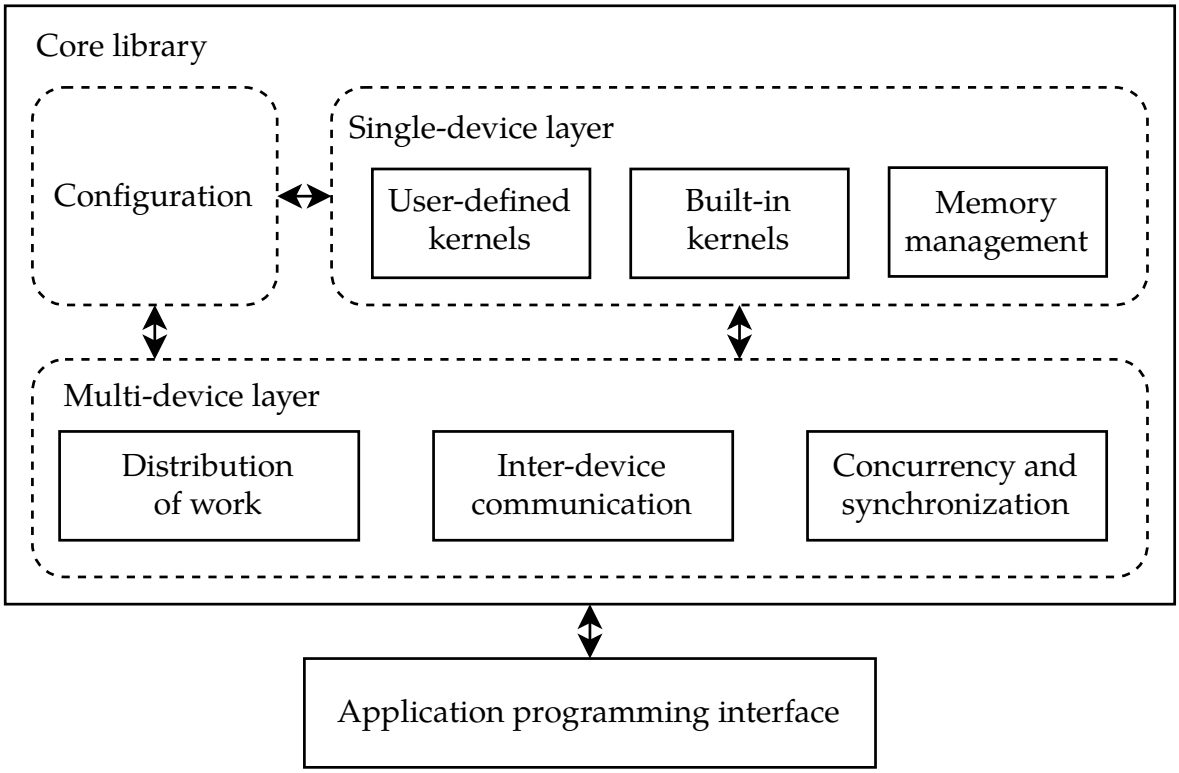


Figure 4.1: Components of the library created for this work.

License v3.0 and freely available at an online repository [88].

4.1 Library architecture and API

We present the architecture of the library and the application programming interface in this section. The library is used to manage the resources of multiple devices in one compute node. In this work, we focus on the aspects of using a single device for the computations and leave discussion on multiple devices out of scope. The library is divided into single- and multi-device layers, as shown in Figure 4.1.

Functions defined on the single- and multi-device layers have access to a configuration header, which declares common datatypes, such as the precision of floating-point numbers, the number of fields and constants used in the computations, and the data layout used with the fields. Functions local to some specific device, such as memory management and kernel calls, are defined on the single-device layer, while synchronization, communication, concurrency and distribution of the grid among devices within the node are managed on the multi-device layer. The benefit of dividing the library into single- and multi-device layers is, that single-device implementations can be modified, for example by changing the memory layout to

improve performance, without having to modify code on the multi-device layer.

The library itself is single-threaded, but kernels can be executed asynchronously with the host code and synchronized using CUDA streams and events. Streams and events are created on the single-device layer while the concurrent execution of the kernels is managed on the multi-device layer by supplying an appropriate stream to single-device functions. The benefits of using concurrency are two-fold. First, two or more independent kernels can be executed simultaneously on a single device if there are enough available resources. This is a form of task-parallelism, where tasks are distributed across the SIMT processors of the GPU and in this sense, the tasks are executed in parallel [3, 25]. Second, when distributing the work among multiple devices and synchronizing the execution with CUDA events, the overhead caused by communicating the ghost zones among devices can potentially be hidden if communication is performed simultaneously with computation. For example, the values in the ghost zones are only used to compute stencils near the boundaries, while any other stencil can be processed with data already local to the device, which could be done simultaneously with the communication of the ghost zones.

The library is used via an application programming interface (API), which defines all the necessary functions for transferring data fields across host and device memories, and controlling execution on the device. For controlling execution, the API provides functions for synchronization, which go through the multi-device layer and where the work is distributed among devices in the node in addition to the built-in and user-defined functions. Listings 4.2 and 4.1 show the functions provided with the interface and an example of a simulation loop. All calls to the user-defined or built-in kernels are non-blocking.

The library is built around a kernel, which applies stencil operations on a set of vertices. This kernel is generated using the domain-specific language discussed in the next section. In this work, we use this kernel to perform integration substeps to advance the state of an electrically conducting fluid. We also provide two types of built-in functions with the library. First, we provide kernels for performing reductions with scalar and vector fields, and for applying periodic boundary conditions on the ghost zone of a field. Second, we provide device functions, which can be called within user-defined kernels. At the time of writing, we provide built-in functions for computing first-, second-order and cross partial derivatives with respect to all axes, several functions for basic linear algebra, such as the dot product and some more complex functions, such as the curl operation. In Section 4.2, we present the built-in functions in a format in which they can be called within our domain-specific language.

```c
#include <astaroth.h>

#include "host_utilities.h" // load_config, acmesh_create, ac_mesh_destroy, ...

int
main(void)
{
    /* Load the configuration from a file */
    AcMeshInfo mesh_info;
    load_config(&mesh_info);

    /* Compute the initial condition */
    AcMesh* mesh = acmesh_create(mesh_info);
    acmesh_init_to(INIT_TYPE_RANDOM, mesh);

    /* Initialize the GPUs and load the mesh */
    acInit(mesh_info);
    acLoad(*mesh);

    /* Step the simulation */
    while (true) {
        const AcReal umax = acReduceVec(RTYPE_MAX, VTXBUF_UUX, VTXBUF_UUY, VTXBUF_UUZ);
        const AcReal dt   = adaptive_timestep(umax, mesh_info);
        acIntegrate(dt);
    }
    acSynchronize();

    /* Deallocate memory */
    acQuit();
    acmesh_destroy(mesh);
    return 0;
}
```

Listing 4.1: An example of a simulation loop, where integration and reductions are executed on a compute node.

4.2 Domain-specific language

In this section, we present a novel domain-specific language (DSL) designed to provide users with a simplified way of writing efficient kernels for tasks involving stencil operations. The benefits of creating a DSL are twofold. First, scientists using the language can focus on developing solvers and mathematical models using a high-level language while still achieving performance close to handwritten code. Second, procedures written in the DSL are decoupled from implementation. This allows us to extend the DSL compiler to generate optimized code for different architectures, or to generate an intermediate representation that can be further compiled to optimized cross-platform code by compiler suites such as Delite [18] or Lift [17]. The drawback is an added layer of complexity, as the translation from the DSL to CUDA is not obvious.

Next, we discuss the features of the DSL and justify our design decisions. Our first consideration was the programming paradigm. We considered functional style, because it is an intuitive way to represent functions and has been adopted by many

related DSLs [20, 21, 89] and intermediate representations [18, 57], but ultimately chose procedural style because it is well-known and widely used in science and industry [13, 19, 76, 90]. C-like programs are straightforward to compile to CUDA, as little changes are required when translating to and from the intermediate representation of the code. This simplifies compiler development and potentially makes debugging easier.

We chose dataflow programming as the programming model. In this model, a chain of functions is executed on a stream of data, where the output of a function is passed to the next. In our case, the chain of functions defined within a kernel are executed on all or a subset of vertices belonging to the computational domain. Dataflow programming provides a natural way to express stages and interactions in the logical graphics pipeline, which is why shading languages are also based on this model [25]. Graphics processing units are designed to excel in tasks expressed in this fashion.

Our language is limited to writing GPU programs and we do not provide means to manage memory outside of the scope of kernels. The library discussed in Section 4.1 is designed to provide all the necessary functions for managing host and device memory among other operations outside kernels. We separated our DSL into two distinct, but closely related languages, analogous to how shader stages are expressed in shading languages, such as GLSL [76]. These distinct languages are used to specify computations for the stencil assembly and processing stages. These stages are defined in separate compilation units. We opted to separate the stages in order to simplify the language, because if all the stages were defined in the same compilation unit, we would have had to introduce additional keywords to determine which functions are available at which stage, and how the functions should be generated during compilation.

The syntax of our language is close to C and C++. In addition to common keywords found in C-like languages, such as `int`, we added the following keywords, which enable us to translate a high-level representation of the stencil pipeline into an efficient CUDA kernel.

```
Scalar  Vector       Matrix
Kernel  Preprocessed uniform
in      out          int3
```

`Kernel` and `Preprocessed` are function qualifiers, while the rest of the keywords are used to qualify and specify variables. Next, we discuss these keywords in detail.

First, a function qualified as `Kernel` is the starting point of the stencil pipeline. The

keyword `Preprocessed` expresses which functions should be evaluated immediately after entering a `Kernel` function. Because the efficiency of the algorithm described in the next section is based on preprocessing stencil operations with small data sets at a time, it is critical to know which functions should be evaluated, such that the result can be stored in local memory for later use. The `Preprocessed` function qualifier may only be used during the stencil assembly stage to enforce that all reads from input arrays are completed and the preprocessed functions are evaluated before entering the stencil processing stage. In future work, the need for the `Preprocessed` function qualifier can be eliminated if the functions most suitable for preprocessing are chosen using heuristics instead of relying on the user to supply the keyword. In this case, we would have to keep track of global memory references during compilation, and balance the usage of local memory with redundant transactions from the device memory. For this work, we leave further discussion on such heuristics out of scope.

For the stencil assembly and processing stages, we introduced qualifiers for input and output arrays, `in` and `out`, respectively. These are analogous to the `in` and `out` qualifiers in shading languages. We use these qualifiers to avoid race conditions, and to determine which arrays can be read through the read-only cache. A race condition happens when the result of a multithreaded program depends on the execution order of its threads, such as when two threads try to read from and write to the same memory location concurrently. We enforce that input arrays are used only for reading, such that they can be read through the read-only cache optimized for spatially local memory fetches, and a thread may read from any memory location without causing a race condition. With output arrays, we require that each thread may only access a unique memory address. The syntax for accessing arrays in our language is similar to C and related languages, with the exception that three-dimensional arrays are accessed using the syntax `array[i, j, k]`, where i, j and k are the coordinates of the vertex being processed. We provide the coordinates of the currently processed vertex with a built-in variable `vertexIdx`, which is of type `int3`. `int3` is a structure composed of three integers. Arrays can also be accessed without specifying the index, in which case the index (`vertexIdx.x`, `vertexIdx.y`, `vertexIdx.z`) is implicitly assumed.

For convenience, we provide three built-in data types for arithmetic, `Scalar`, `Vector` and `Matrix`, and a subset of basic linear algebra subroutines as built-in functions, such as dot product and matrix multiplication. Precision of these data types is not specified with the DSL and may be freely selected by the implementer. In our library, we declare these data types either as single- or double-precision floating-point numbers. Finally, the keyword `uniform` is used to determine which global variables

Table 4.1: The basic built-in functions and variables provided by the library. We refer the reader to [88] for a full list of the built-in functions.

Built-in function or variable	**Type**	**Explanation**
`vertexIdx`	`int3`	Index of the currently processed vertex
`derx(int3 index, in Scalar field)`	`Scalar`	First derivative with respect to the x axis
`derxx(int3 index, in Scalar field)`	`Scalar`	Second derivative with respect to the x axis
`derxy(int3 index, in Scalar field)`	`Scalar`	Cross partial derivative with respect to the x and y axes
`dot(Vector a, Vector b)`	`Scalar`	Dot product $\mathbf{a} \cdot \mathbf{b}$
`length(Vector a)`	`Scalar`	Vector length $\sqrt{a_1^2 + a_2^2 + a_3^2}$
`cross(Vector a, Vector b)`	`Vector`	Cross product $\mathbf{a} \times \mathbf{b}$
`mul(Matrix A, Vector x)`	`Vector`	Matrix-vector product $\mathbf{Ax}$

should be in constant memory of the device. The full list of keywords accepted in the language is shown in Appendix A. The build-in functions provided by our library are shown in Table 4.1. In addition, we provide the function `rk3(out T field_out, in T field_in, T rate_of_change, Scalar dt)` for computing integration steps, where `T` is either a `Scalar` or `Vector`. `rk3` computes the third-order Runge-Kutta integration step using Williamson's coefficients $\alpha_2 = -5/9$, $\alpha_3 = -153/128$, $\beta_1 = 1/3$, $\beta_2 = 15/16$, and $\beta_3 = 8/15$ [19, 59].

Before introducing the syntax of our DSL, let us review the terminology. First, *identifier* is the name of some variable or function. A variable or function is associated with information of its type and storage class. These associations are established with a *declaration*. A *type specifier* specifies the type of a variable, or the return type of a function. A *type qualifier* specifies the storage class for some variable or function. An *expression* consists of at least one identifier or a constant, and an optional operator. The operator can either be an unary operator, in which case no additional operands are needed, or a binary operator, where the operand on its left-hand side is another expression. A *statement* expresses an action to be executed, such as a declaration, an assignment or a return statement. Finally, a *definition* is a statement, where an identifier is declared and assigned a value given by some expression. An example of a definition in C syntax is given below.

$$\overbrace{\underbrace{\text{<type qualifier> <type specifier> <identifier>}}_{\text{declaration}} = \overbrace{\text{<expression> <operator> <operand>}}^{\text{expression}}}^{\text{definition statement}}$$

Next, we specify the syntax of our DSL using the Backus-Naur form (BNF), which is a commonly-used notation for describing context-free grammars [91, 92]. Context-free grammars are defined using productions, which map nonterminal symbols to other nonterminal or terminal symbols [92]. Productions are expressed in the form $A \rightarrow \alpha|\beta$, where A is a nonterminal that is rewritten either as α or β. If there is no production, where α is on the left-hand side, then α is called a terminal symbol. A grammar is context-free, if each rule maps only one symbol from the left-hand side to some nonterminal or terminal [91, 92]. For example, the above statement can be parsed with the following grammar in BNF.

```
definition         → declaration = expression
declaration        → type-definition identifier
type-definition    → type-qualifier
                   | type-qualifier type-specifier
expression         → unary-expression
                   | expression binary-operator unary-expression
unary-expression   → postfix-expression
                   | unary-operator postfix-expression
postfix-expression → primary-expression
primary-expression → identifier
                   | number
                   | ( expression )
```

Our grammar is an extended subset of C. We refer the reader to Kernighan and Ritchie [93] for a detailed discussion on C grammar. We extended the index notation, such that multidimensional arrays can be accessed with the notation `array[i, j, k]` instead of `array[i][j][k]`, as this requires fewer symbols and does not suggest a specific data layout. We specify this rule by adding the following productions to the grammar shown above.

```
expression-list    → expression
                   | expression-list , expression
                   ;
```

```
postfix-expression → primary-expression
                   | postfix-expression [ expression-list ]
                   ;
```

We deliberately left out some features of C. Notably, we require that the scope of `if`, `else if`, `else`, `while` and `for` statements is explicitly indicated with braces. We also allow increment and decrement by one shorthands ++ and -- to be used only as a prefix, as this removes confusion about the differences between prefix and postfix notations. We do not allow `goto` statements either. The full grammar of the language developed for this work is shown in Appendix B. Implementation of the magnetohydrodynamics solver developed for with work with our DSL is shown in Appendix C.

Listings 4.3 and 4.4 show how the heat equation $\frac{\partial \mathbf{T}}{\partial t} - \alpha \nabla^2 \mathbf{T} = 0$ may be solved with our DSL. With our compiler, these sources are compiled into CUDA code equivalent to that shown in Listings 4.5 and 4.6.

4.3 DSL compiler and code generation

In addition to designing a domain-specific language (DSL), we created source-to-source compiler for generating CUDA kernels from programs written in this language. Our compiler incorporates several phases. First, a high-level representation of a stencil-processing pipeline is given as input to the lexical analyzer, which extracts tokens from the source and passes those to a syntax analyzer. With a syntax analyzer, we construct an abstract syntax tree based on context-free grammar that specifies the language. This abstract syntax tree serves as an intermediate representation of the code. Finally, during the code optimization and generation phase, we traverse the syntax tree while maintaining a symbol table and generate a CUDA kernel from the intermediate representation. The generated kernel is then embedded in our library and compiled further to machine-specific code with the NVIDIA CUDA compiler. Figure 4.2 shows the compilation phases of our DSL.

Next, we discuss the technical details of the generated kernel. In this work, we use the generated kernel to compute integration substeps. When designing the algorithm generated with the DSL compiler, our primary consideration was to ensure that the generated code could be used to perform computations with stencils of various shapes relatively efficiently without having to tune the kernel by hand for different microarchitectures. As discussed in Chapter 1, it is often the case that low-level optimizations do not carry over hardware generations. With

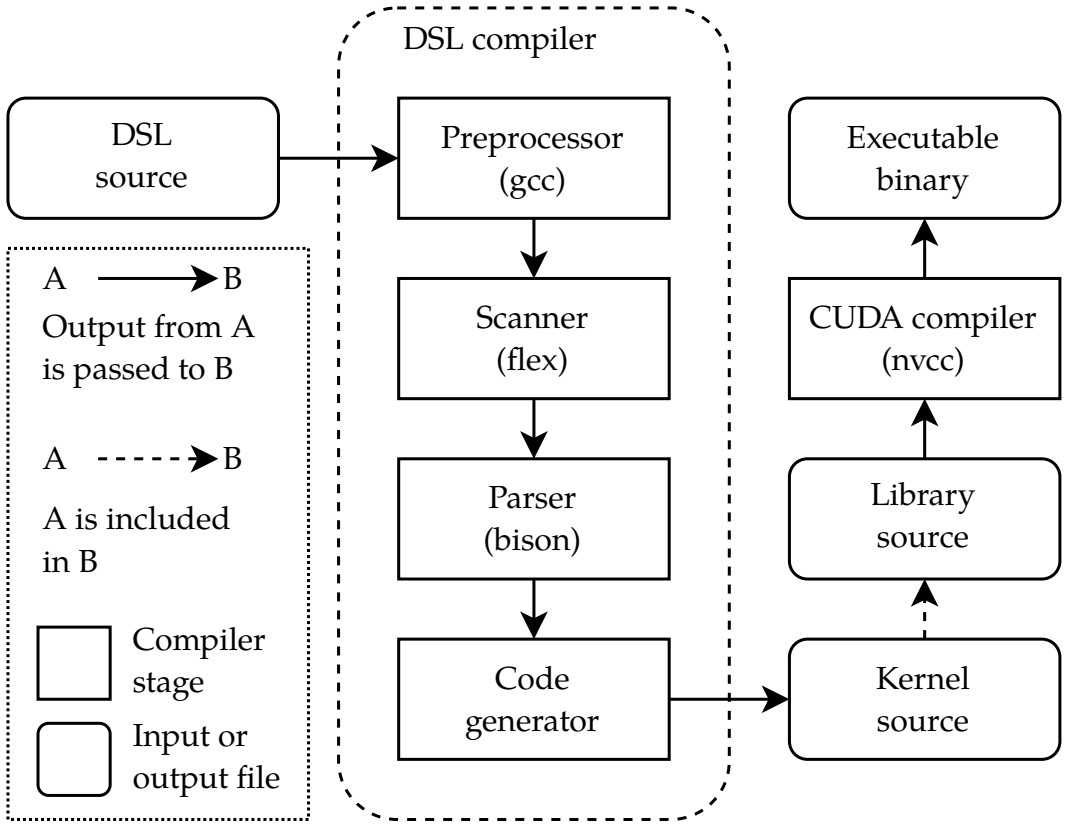


Figure 4.2: The compilation phases used to translate kernels written in our DSL to CUDA. The kernel is then included in our library, which is compiled into an executable binary with the NVIDIA CUDA compiler.

each GPU microarchitecture, new optimization techniques and hardware features are introduced, which may not be available on older machines. By the time a rigorously optimized implementation on one architecture has been completed, it may be that a significantly faster device requiring a different approach to obtain optimum performance has already been released. Therefore during code generation, we decided to apply only high-level optimizations that are less likely to change across GPU architectures, and rely on the CUDA compiler and an auto-tuning script to optimize the program further.

In stencil computations, if a CUDA thread is assigned for updating a single grid point, then the threads updating neighboring points share a part of the data required for computing the output. We say that the stencils of those neighbors overlap, as shown in Figure 4.3. In the ideal case, all of the shared data used for computing neighboring stencils would be fetched into caches and not evicted, until all threads utilizing that data would have completed their computations. In terms of minimizing redundant accesses to device memory, the most efficient solution would be to cache

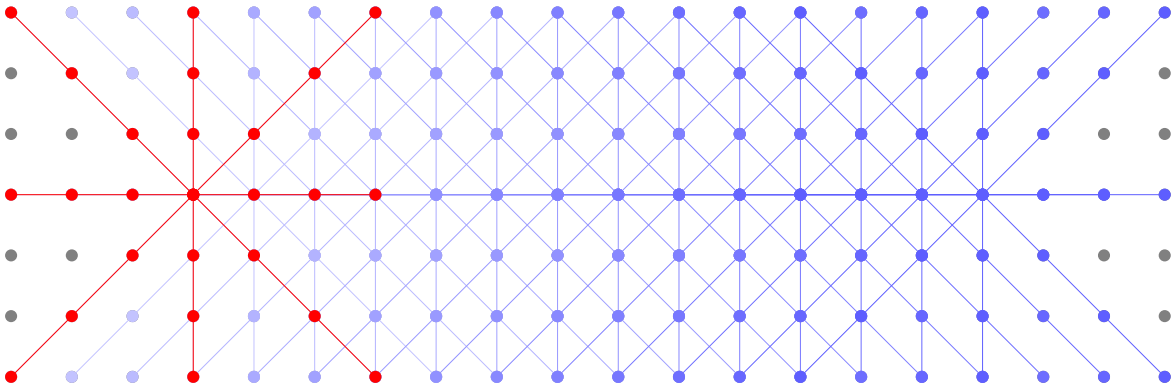


Figure 4.3: Overlapping 25-point stencils.

the entire computational domain for the duration of the kernel as all neighboring stencils overlap to some extent. In practice, the size of the CTA and the amount of data that can be stored in caches is limited. Instead, the computational domain must be decomposed into smaller blocks, where the working set is small enough to fit into caches. This approach is called cache blocking.

As stated in the problem statement in Section 1.3, our solver must support adding new equations and fields, which may change the stencil shape and consequently change the optimal way to block the data. As finding a performance-portable way to cache arbitrary-sized stencils programmatically with shared memory is a major undertaking, we opted instead to rely on *implicit caching*. We use this term to refer the type of caching, that is carried out by the hardware and drivers. For example, we consider actions, such as selecting registers for reusing and utilizing cache replacement policies, prefetching and other techniques to store data in L1, L2 and texture caches, to be types of implicit caching. In contrast, we use the term *explicit caching* to refer to explicitly managing the user-programmable portion of the on-chip cache allocated as shared memory. While generally relying on implicit caching on GPUs is not recommended because of a large number of threads competing for a shared cache [94], which is likely to lead to thrashing, in Chapter 5, we show that in our case implicit caching works surprisingly well. The potential reason for this is that as we increase resources allocated per thread to mitigate register spilling, this limits the number of threads that can be multithreaded on a SIMT processor, which in turn reduces competition for the L1 cache. The potential downside is, that if each warp of a CTA operates concurrently on a different field, then the working set of a CTA is unlikely to fit into L1, which would lead to higher conflict and capacity miss rates.

However, if the warps of a CTA execute instructions in a relatively synchronized

fashion, such that each warp would access nearby locations in a single array in quick succession, then the data required by those threads would more likely be resident in the cache. Devices based on the Pascal microarchitecture have been suggested to use least-recently-used as their caching policy [26], which in this case would hold many of the vertices accessed by multiple threads in the L1 cache. In addition, we have decomposed the problem in a way, where each thread updates one vertex and a warp operates on 32 contiguous vertices. As cache lines are replaced in segments of 128 bytes [26, 67, 74], a load instruction from one thread triggers the transfer of a cache line containing data required also by other threads in that warp. Recall also that the warp schedulers choose a warp for execution from all warps that are ready to run [25]. We suspect that this contributes to keeping the warps of a CTA running in a relatively synchronized fashion, as threads accessing uncached data stall for 230–370 cycles [26, 62], while threads which have their working set in registers and L1 may execute instructions in the meantime. To support this hypothesis, previous work suggests that L1 hit latency with the Pascal microarchitectures is 82 cycles, while the execution latency of floating point instructions ranges from 6 to 14 cycles [26]. In addition, as each of the stream processors can be issued one instruction per clock cycle [25, 75], several instructions could potentially be executed before a cache line replacement triggered by a warp further ahead in execution is completed. However, to our knowledge the exact details on how cache lines are replaced on GPU hardware are not publicly documented.

We also did experiments with shared memory but were not able to find a solution that was noticeably faster. With large stencils, say, 55 points where each point comprises 32 bytes of data, large amounts of shared memory are likely needed to achieve a satisfactory reuse ratio. However, using large amounts of shared memory decreases the occupancy of the kernel and requires a synchronization, which stalls the threads until the data has been fetched to caches. As latencies are not hidden in this case by multithreading, and memory and arithmetic operations cannot be executed while waiting for the synchronization, such kernel is likely to be bound by latency. We leave shared memory out of scope for the rest of the work. For a more detailed discussion on issues with using shared memory with large stencils, we refer the reader to [44].

Our second major consideration was to focus on employing instruction-level parallelism to hide latencies instead relying purely on fine-grained multithreading. As arithmetic performance in terms of floating-point operations per byte is much higher than the rate bytes can be transferred from the device memory [26], kernels

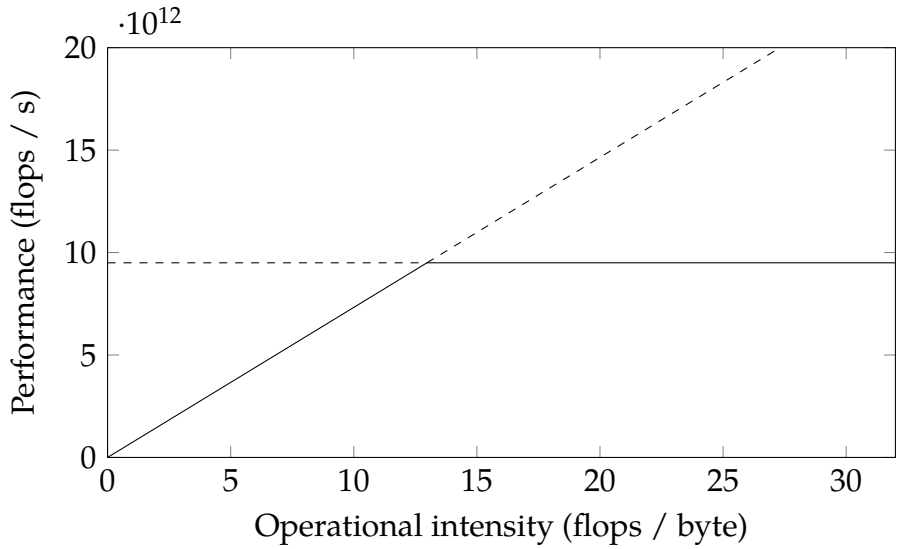


Figure 4.4: The roofline performance of a Tesla P100 PCIe GPU. The theoretical maximum performance was plotted with Williams' roofline model given by $\min(W, I \cdot Q)$ [95], where $W = 9.5 \cdot 10^{12}$ flops / s and $0.73 \cdot 10^{12}$ bytes / s [26]. The performance of tasks which have an operational intensity of $I = W/Q \approx 13$ are bound by both memory bandwidth and compute performance. In tasks where $I < 13$ the performance is bound by memory bandwidth.

with low- to moderate operational intensity[1] are likely to be bound by bandwidth. Operational intensity is calculated as the ratio of operations per bytes transferred $I = W/Q$ [95]. Figure 4.4 shows the roofline model for a Tesla P100 PCIe GPU used in this work.

It is difficult to give an exact value for the operational intensity of the generated integration kernel because of the large amount of arithmetic involved. Instead, we measured the number of floating-point operations performed and bytes transferred experimentally, and estimated the operational intensity of computing integration substeps to be 2.7–6.9. We discuss this experiment in detail in Section 5.2. In problems where operational intensity is less than what is needed to utilize the compute units, hiding latencies by running a greater number of threads per processor is only effective for reaching the maximum bandwidth, after which caches must be utilized to improve performance. The low operational intensity of the kernel indicates a strong need for caching in order to reduce pressure to device memory. Additional methods can also be used to reduce pressure on the memory bus, such as compressing the data before transferring it over the network [96] or on-chip, but

[1]Less than 13 floating-point operations per byte transferred on a Tesla P100, given a peak performance of $W = 9.5 \cdot 10^{12}$ floating-point operations per second, when counting the fused multiply-add as one operation, and a memory bandwidth of $Q = 682$ GiB / s.

we leave these techniques out of scope.

In our kernel, we rely on fewer threads to supply the necessary instructions to hide operational and memory access latencies instead of focusing on increasing the number of threads being multithreaded on a SIMT processor. However, the number of simultaneously executed instructions that can be supplied by a single warp is limited [62], which makes the kernel more sensitive to instruction stalls. The primary way to mitigate stalling is to ensure that as much of the data as possible is available in registers and caches when needed. To this end, we increased the register limit per thread to the maximum allowed by the hardware. The primary benefits of this are, that fetching data from registers takes roughly two times fewer clock cycles than from shared memory [62] and six times less cycles than from the L1 cache [26]. A larger register file also mitigates spilling data to caches. In addition to allocating more resources per thread, we inline small functions that are called often, such that additional registers are not used to pass the arguments, and the contents of the function call are visible, such that for example a load-store instruction inside a function could be dispatched simultaneously with an independent arithmetic instruction outside that function. Finally, we strive to ensure that all loops can be unrolled by fixing the number of iterations at compile time, such that stream processors and clock cycles are not wasted on evaluating the branch condition. The drawback of both inlining and unrolling is the increased pressure to the instruction cache.

We leave lower-level optimizations than discussed here out of scope, as performance improvements from these techniques would depend on the device architecture. Examples of such low-level optimizations would be ordering instructions in a way that all available instructions slots of the device would be utilized, or renaming registers in a way that there are no false data dependencies or register bank conflicts. Additionally, we expect the performance to be limited by peer-to-peer and network bandwidth when the library is extended to support computations on multiple nodes in later work, which is why we have left out also some higher-level optimizations. For example, changing indexing within the grid to improve spatial locality of the data or padding the computational domain may reduce the number of memory transaction required in the kernel at the cost of increasing transactions over a network.

Next, we give an overview of the generated kernel. The kernel comprises two programmable stages, shown in Figure 4.5. Examples of the programmable stages written in our DSL are shown in Listings 4.5 and 4.6. The stages are compiled into CUDA headers, which are embedded in a single compilation unit. In addition to the generated pipeline stages, this compilation unit consists of built-in functions, such as the dot product, and the necessary code for calling the integration kernel. Before

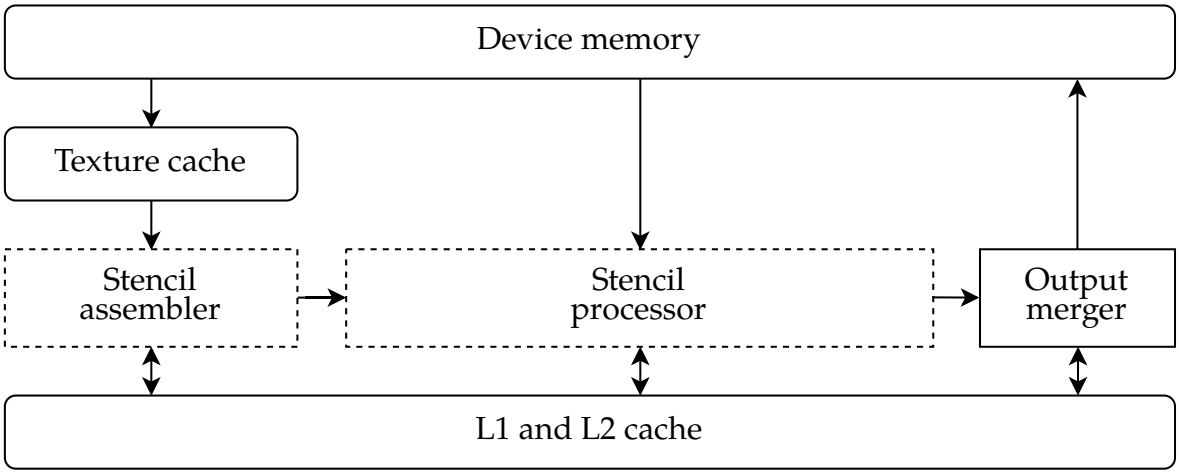


Figure 4.5: A conceptual model of the stencil pipeline generated with our compiler. The stencil assembly and processing stages are programmable with the DSL. During stencil assembly, the data qualified as `in` is read from device memory through the texture cache, preprocessed, and the results of the stencil operations are stored into local memory. The data stored in local memory is then used to solve functions defined in the stencil processing stage. During the output merger stage, the results are written back to device memory. See Figure 2.1 for a comparison with a traditional graphics pipeline.

calling the integration kernel, we decompose the grid to blocks of vertices, which are assigned to the threads of a CTA. Within a CTA, each CUDA thread updates a single grid point. After entering the kernel, the input arrays are read and the stencils are preprocessed using the functions qualified as `Preprocessed` in the DSL source code. The results of `Preprocessed` functions are then stored to a structure held in local memory. After all `Preprocessed` functions have been processed, the execution continues from the code defined for the stencil processing stage. After the stencil processing stage has been completed, the results are written back to device memory.

```
/** Initialize all GPUs in the given node. The mesh, device
    constants are allocated and necessary data for further
    computations is distributed among GPUs */
AcResult acInit(const AcMeshInfo& mesh_info);

/** Decompose the mesh stored in host memory and distribute
    it among GPUs in the node */
AcResult acLoad(const AcMesh& host_mesh);
AcResult acLoadWithOffset(const AcMesh& host_mesh,
                          const int3& start,
                          const int num_vertices);

/** Perform a full integration step, computing boundary
    conditions where needed */
AcResult acIntegrate(const AcReal& dt);

/** Perform an integration substep by calling the kernel
    generated from the domain-specific language */
AcResult acIntegrateStep(const int& isubstep,
                         const AcReal& dt);

/** Compute and communicate ghost zones among GPUs in the
    node */
AcResult acBoundcondStep(void);

/** Perform a scalar reduction of a given type */
AcReal acReduceScal(const ReductionType& rtype,
                    const VertexBufferHandle& handle);

/** Perform a vector reduction of a given type */
AcReal acReduceVec(const ReductionType& rtype,
                   const VertexBufferHandle& arrx,
                   const VertexBufferHandle& arry,
                   const VertexBufferHandle& arrz);

/** Store the mesh from GPU memory to CPU memory */
AcResult acStore(AcMesh* host_mesh);
AcResult acStoreWithOffset(const int3& start,
                           const int num_vertices,
                           AcMesh* host_mesh);

/** Free all allocations on the GPUs and reset */
AcResult acQuit(void);

/** Synchronize all devices in the node */
AcResult acSynchronize(void);
```

Listing 4.2: Description of the application programming interface for accessing the library created for this work. Calls to user-defined and built-in kernels are executed asynchronously with respect to the host code. The interface provides the function `acSynchronize()` for synchronizing all devices used by the library.

```
uniform Scalar inv_dsx, inv_dsy, inv_dsz;

Scalar
second_derivative(Scalar* pencil, Scalar inv_ds)
{
    Scalar coefficients[] = {...};

    #define MID (STENCIL_ORDER / 2)
    Scalar res = coefficients[0] * pencil[MID];

    for (int i = 1; i <= MID; ++i)
        res += coefficients[i] * (pencil[MID + i] + pencil[MID - i]);

    return res * inv_ds * inv_ds;
}

Scalar
derxx(in Scalar field)
{
    Scalar pencil[STENCIL_ORDER + 1];

    for (int offset = 0; offset < STENCIL_ORDER + 1; ++offset)
        pencil[offset] = field[vertexIdx.x + offset - STENCIL_ORDER / 2,
                               vertexIdx.y,
                               vertexIdx.z];

    return second_derivative(pencil, inv_dsx);
}

Preprocessed Scalar
laplacian(in Scalar field)
{
    return derxx(field) + deryy(field) + derzz(field);
}
```

Listing 4.3: Sample code for generating the stencil assembly stage.

```
uniform Scalar alpha;

Vector
laplacian(in Vector T)
{
    return (Vector){laplacian(T.x), laplacian(T.y), laplacian(T.z)};
}

Vector
heat_equation(in Vector T)
{
    return alpha * laplacian(T);
}

in  Vector T_in  = (int3){0, 1, 2};
out Vector T_out = (int3){0, 1, 2};

Kernel
solve(Scalar dt)
{
    T_out = T_in + heat_equation(T_in) * dt;
}
```

Listing 4.4: Sample code for generating the stencil processing stage.

```
__constant__ float inv_dsx, inv_dsy, inv_dsz;

typedef struct {
    float value;
    float laplacian
} PreprocessedVertex;

typedef struct {
    PreprocessedVertex x;
    PreprocessedVertex y;
    PreprocessedVertex z;
} PreprocessedVertex3;

static __device__ __forceinline__ float
second_derivative(const float* __restrict__ pencil, const float inv_ds)
{
    const float coefficients[] = {...};

    #define MID (STENCIL_ORDER / 2)
    float res    = coefficients[0] * pencil[MID];

    #pragma unroll
    for (int i = 1; i <= MID; ++i)
        res += coefficients[i] * (pencil[MID + i] + pencil[MID - i]);

    return res * inv_ds * inv_ds;
}

static __device__ __forceinline__ float
derxx(const int3 vertexIdx, const float* __restrict__ arr)
{
    float pencil[STENCIL_ORDER + 1];
    #pragma unroll
    for (int offset = 0; offset < STENCIL_ORDER + 1; ++offset)
        pencil[offset] = arr[IDX(vertexIdx.x + offset - STENCIL_ORDER / 2,
                                 vertexIdx.y,
                                 vertexIdx.z)];

    return second_derivative(pencil, DCONST_REAL(AC_inv_dsx));
}

static __device__ __forceinline__ PreprocessedVertex
stencil_assembly(const int3 vertexIdx,
                 const float* __restrict__ arr)
{
    PreprocessedVertex vtx;

    vtx.value = arr[IDX(vertexIdx)];
    vtx.laplacian = (float3){derxx(vertexIdx, arr) +
                             deryy(vertexIdx, arr) +
                             derzz(vertexIdx, arr)};

    return vtx;
}

static __device__ __forceinline__ PreprocessedVertex3
stencil_assembly(const int3 vertexIdx,
                 const float* __restrict__ arrx,
                 const float* __restrict__ arry,
                 const float* __restrict__ arrz)
{
    PreprocessedVertex3 vtx;

    vtx.x = stencil_assembly(vertexIdx, arrx);
    vtx.y = stencil_assembly(vertexIdx, arry);
    vtx.z = stencil_assembly(vertexIdx, arrz);

    return vtx;
}
```

Listing 4.5: The stencil assembly stage of the stencil pipeline.

```
__constant__ float alpha;

static __device__ __forceinline__ float3
laplacian(const PreprocessedVertex3& vtx)
{
    return (float3){vtx.x.laplacian, vtx.y.laplacian, vtx.z.laplacian};
}

static __device__ __forceinline__ float3
heat_equation(const PreprocessedVertex3& T)
{
    return alpha * laplacian(T);
}

static __device__ __forceinline__ float3
stencil_process(const PreprocessedVertex3& T, const float dt)
{
    return T.value + heat_equation(T) * dt;
}


template <int step_number>
static __global__ void
__launch_bounds__(RK_THREADBLOCK_SIZE, RK_LAUNCH_BOUND_MIN_BLOCKS)
solve(const int3 start, const int3 end, const float dt,
      VertexBufferArray buffer)
{
    const int3 vertexIdx = (int3) {
        threadIdx.x + blockIdx.x * blockDim.x + start.x,
        threadIdx.y + blockIdx.y * blockDim.y + start.y,
        threadIdx.z + blockIdx.z * blockDim.z + start.z
    };

    if (vertexIdx.x >= end.x ||
        vertexIdx.y >= end.y ||
        vertexIdx.z >= end.z) {
            return;
    }

    const PreprocessedVertex3 T = stencil_assembly(vertexIdx,
                                                   buffer.in[0],
                                                   buffer.in[1],
                                                   buffer.in[2]);
    const float3 result = stencil_process(T, dt);

    buffer.out[0][IDX(vertexIdx)] = result[0];
    buffer.out[1][IDX(vertexIdx)] = result[1];
    buffer.out[2][IDX(vertexIdx)] = result[2];
}
```

Listing 4.6: The processing stage of the stencil pipeline.

Chapter 5

Results

In this chapter, we present the results on how well our library satisfies the requirements set in Section 1.3. We evaluated the library in three tests. First in Section 5.1, we verify that all functions provided by the library and our solver give results that are within expectable error bounds. We show that the difference in results when comparing our GPU-solver using 32-bit and 64-bit precision with a sequential CPU-solver using 80-bit precision is within the reasonable bounds of fixed-precision arithmetic. Second in Section 5.2, we measure the hardware utilization of our integration kernel and compare the performance with a conservative lower bound for the running time, that could theoretically be achieved with an optimal algorithm. Finally in Section 5.3, we compare the running time of our solver with a widely-used multiphysics code optimized for high-performance computing on multiple CPUs.

We ran all tests discussed in this chapter with the following hardware. The CPU tests were run on an Apollo 6000 XL230a Gen.9 server blade hosting two 22 nm Intel Xeon E5-2690 v3 processors running at 2.6 GHz. Attached to the server blade were eight 16 GiB fourth-generation double-data rate (DDR4) memory modules, running at a clock rate of 2133 MHz. The memory modules operated with error-correcting codes (ECC) on and provided a theoretical bandwidth of 64 GiB/s in total via four memory channels [7]. One Intel Xeon E5-2690 v3 hosted 12 cores, which shared a 30 MiB L3 cache. Each core had dedicated 32-KiB L1 and 256-KiB L2 caches. The GPU tests were run on a Dell PowerEdge C4130 server blade housing two 14 nm 14-core Intel Xeon E5-2680 v4 processors running at 2.4 GHz, eight 64 GiB modules of DDR4 synchronous dynamic random-access memory (SDRAM) and four NVIDIA GP100GL Tesla P100 PCIe 16 GiB GPUs running at 1.3 GHz. These GPUs were connected in pairs to each CPU via a PCIe 3.0 bus. Each GPU contained 16 GiB of second-generation high-bandwidth memory (HBM2) running at 715 MHz,

providing a theoretical bandwidth of 682 GiB/s via a 4096-bit wide memory bus. Error-correcting code was enabled in all tests. The P100 GPU consisted of 56 SIMT processors, where each SIMT processor had 64 single-precision, and 32 double-precision processing units. The SIMT processors had a shared L2 cache of 4096 KiB [8] and each processor had a dedicated 24 KiB L1 data cache and 64 KiB of shared memory [26]. A single cooperative thread array could be assigned a maximum of 48 KiB of shared memory.

5.1 Verification

For our first test, we created a sequential CPU solver capable of solving the equations described in Chapter 3 and used it to generate a model solution. Then we compared the model solution to the output given by our GPU solver. This comparison was done to confirm that there are no issues with the parallelisation scheme and to analyze the error of performing computations with finite precision. However, making a comparison between floating-point numbers is not straightforward, so we give a short introduction to the subject and discuss the issues before detailing the methods used for the comparison. In the following discussion, we use the term *model* to refer to the solution produced by our sequential solver, which performed the operations with 80-bit extended precision. The term *candidate* is used to refer to the solution produced by our GPU solver in either 32-bit or 64-bit floating-point precision.

While modern CPUs and GPUs conform to the IEEE 754-2008 [68] standard for floating-point arithmetic [8, 24], this does not guarantee that the results of the same program are identical on all IEEE 754-compliant systems [97]. Even though CUDA and GPUs are stated to be IEEE 754-2008 compliant, only basic arithmetic functions, such as multiplication, division, fused multiply-add and square root are defined in the standard [68, 98]. Trigonometric functions for example are not specified [68, 98]. The machine code for CPUs and GPUs are also based on different instruction sets, and optimizations performed during compilation may change the number of rounding steps performed in the program. For example, the compiler might choose to use the fused multiply-add instruction, which performs a multiplication and addition with one rounding step instead of two. This issue is not unique to CPUs and GPUs, but rather the consequence of the fact that floating-point arithmetic is not necessarily associative, that is, $a + (b + c)$ is not always equal to $(a + b) + c$ [99].

When comparing floating-point numbers, the expected error depends on the magnitude of the inputs and the computations performed with the input [99, 100].

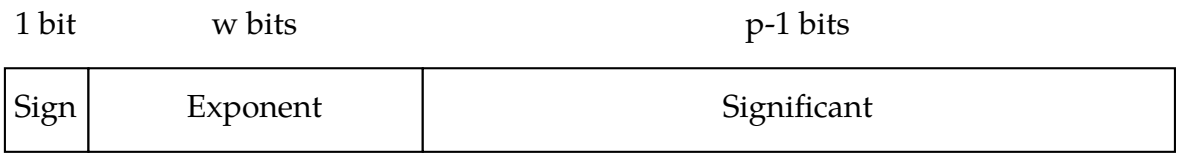


Figure 5.1: IEEE 754-2008 format for 32- and 64-bit binary floating-point numbers. For single-precision, $w_s = 8$ and $p_s = 24$, while $w_d = 11$ and $p_d = 53$ for double precision [68].

To our knowledge, there is no single method for comparing floating-point numbers that would be useful in all cases. Rather, the comparison subroutine has to be tuned to accept differences depending on the computations performed and the expected precision of the results. Floating-point errors are commonly expressed in terms of the relative error with respect to some machine epsilon ϵ or absolute error in terms of units in the last place (ulps) [98–100]. In this work, we analyze the error in terms of ulps using the definition by Higham [101] but in some cases refer also to the machine epsilon ϵ, which is the distance from 1.0 to the next larger floating-point number [99, 101]. The machine epsilon ϵ is given by

$$\epsilon = \beta^{-(p-1)} \tag{5.1}$$

for normal numbers, where β is the base and p the precision of the significant [100]. A floating-point number is normal if its exponent is large enough, such that the leading digit of the significant expressed in binary can be set to one [98]. For 32-bit binary floating-point numbers conforming to the IEEE 754-2008 standard the smallest normal number is $\beta^{e_{\min}} = \beta^{1-e_{\max}} = 2^{-126}$, where $e_{\min}$ and $e_{\max}$ are the minimum and maximum exponents the number can represent, respectively [68].

Higham defines ulps as the interval between two finite floating-point numbers [101, 102]. Given some candidate value c and a model value $m \neq 0$, units in the last place are defined as

$$ulp(m) = \beta^{e-(p-1)} \, . \tag{5.2}$$

Here e is the precision of the exponent of m [101, 102]. In our tests, we calculated e as

$$e = \lfloor \log_2 |m| \rfloor \, . \tag{5.3}$$

Figure 5.1 shows the format of 32- and 64-bit binary floating-point numbers specified in the IEEE 754-2008 standard. For binary 32- and 64-bit floating-point numbers $\beta = 2$, and $p_s = 24$ and $p_d = 53$ for single and double precision, respectively [68]. As the

IEEE 754-2008 standard states that all operations must be correctly rounded [68, 100], the absolute error of a single arithmetic operation should be at most $\frac{1}{2}$ ulps [99], that is

$$|c - m| \leq \frac{1}{2} ulp(m) \, . \tag{5.4}$$

Next, we describe the test cases. In the first initial condition named *random*, we initialized all fields used in the simulation to random numbers in range $[10^{-2}, 10^{-2}]$. If larger numbers were used, the test case would become unstable with both the model and candidate solutions. In the test *X-wave*, we initialized the x-component of velocity to a sinusoidal wave with respect to the y-coordinate in the grid. The value was given by $2\sin(\pi j/m_y) - 1$, where j is the index of the vertex in the y direction and m_y is the total number of vertices also in the y direction. Fields other than the velocity field were initialized to random values in range $[10^{-2}, 10^{-2}]$. In the test *radial explosion*, we created a radial Gaussian velocity profile originating from the center of the grid. Other fields that velocity were initialized to a constant value of 1. We refer the reader to the subroutine `gaussian_radial_explosion` in the code repository [88] for a detailed definition. In the final test, *ABC-flow*, we initialized the grid to the Arnold-Beltrami-Childress flow given by [103]

$$\dot{x} = A \sin z + C \cos y \tag{5.5}$$

$$\dot{y} = B \sin x + A \cos z \tag{5.6}$$

$$\dot{z} = C \sin y + B \cos x \, , \tag{5.7}$$

where we set $A = B = C = 1$. We initialized the velocity as follows, where $k_u = 32$, (O_x, O_y, O_z) is the position of the center of the computational domain, and $u_x(\delta_i, \delta_j, \delta_k)$ is the state of the x component of the velocity at position $(\delta_i, \delta_j, \delta_k)$

$$u_x(\delta_i, \delta_j, \delta_k) = \sin[k_u(\delta_z - O_z)] + \cos[k_u(\delta_y - O_y)] \tag{5.8}$$

$$u_y(\delta_i, \delta_j, \delta_k) = \sin[k_u(\delta_x - O_x)] + \cos[k_u(\delta_z - O_z)] \tag{5.9}$$

$$u_z(\delta_i, \delta_j, \delta_k) = \sin[k_u(\delta_y - O_y)] + \cos[k_u(\delta_x - O_x)] \, . \tag{5.10}$$

As in the *X-wave* test case, we initialized the values of all fields, except the velocity field, to random values in range $[10^{-2}, 10^{-2}]$.

Recall that a full integration steps involves computing integration substeps and boundary conditions. During an integration substep, stencil operations are performed to solve partial differential equations, and during the boundary condition step, data from the computational domain is copied into the ghost zones according

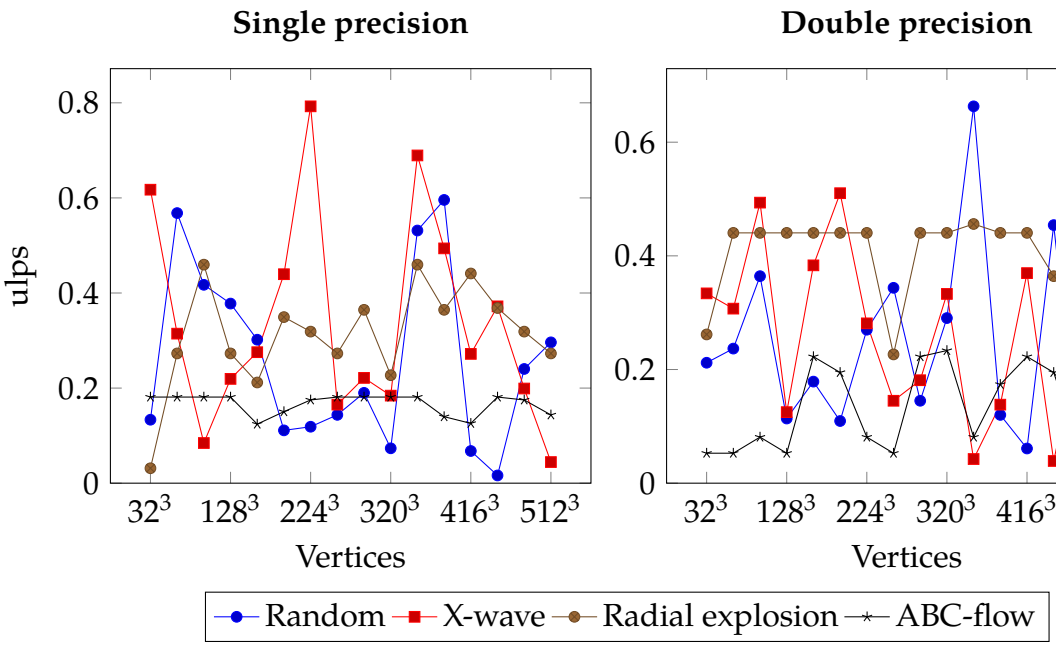


Figure 5.2: Floating-point error for vector reductions using different grid dimensions. The velocity field $\mathbf{u} = (u_x, u_y, u_z)$ was searched for the maximum length of the velocity vector. Results are expressed in terms of machine epsilons for binary 32- and 64-bit floating point numbers specified by the IEEE 754-2008 standard [68]. A single ulp was equal to $\beta^{e-(p-1)}$ as shown in Equation 5.2.

to some boundary conditions, in our case, the periodic boundary conditions. In our tests, given the dimensions of the computational domain n_x, n_y and n_z, the data at index (i, j, k) in the ghost zones were copied from the data at index ($i \mod n_x$, $j \mod n_y$, $k \mod n_z$) in the computational domain. The periodic boundary conditions were applied to all eight fields used in the computations. As no arithmetic was performed and the input data for the model functions had the same precision as the candidate functions solved on the GPU, the error from computing boundary conditions in all test cases was exactly zero.

As a part of computing diagnostics and for verifying other functions, we searched a scalar field for the maximum value. As values in the field are searched and reduced to a result based on some condition, this type of function is called a reduction. We tested scalar reductions in all test cases described earlier by taking the maximum of the field u_x. Likewise for the boundary conditions, the error for scalar reductions was exactly zero. We also searched the velocity field $\mathbf{u} = (u_x, u_y, u_z)$ for the maximum length of the velocity vector, given by $\sqrt{u_x^2 + u_y^2 + u_z^2}$. The error of vector reduction is shown in Figure 5.2.

Finally, we compared the results of a full integration step with the model solution.

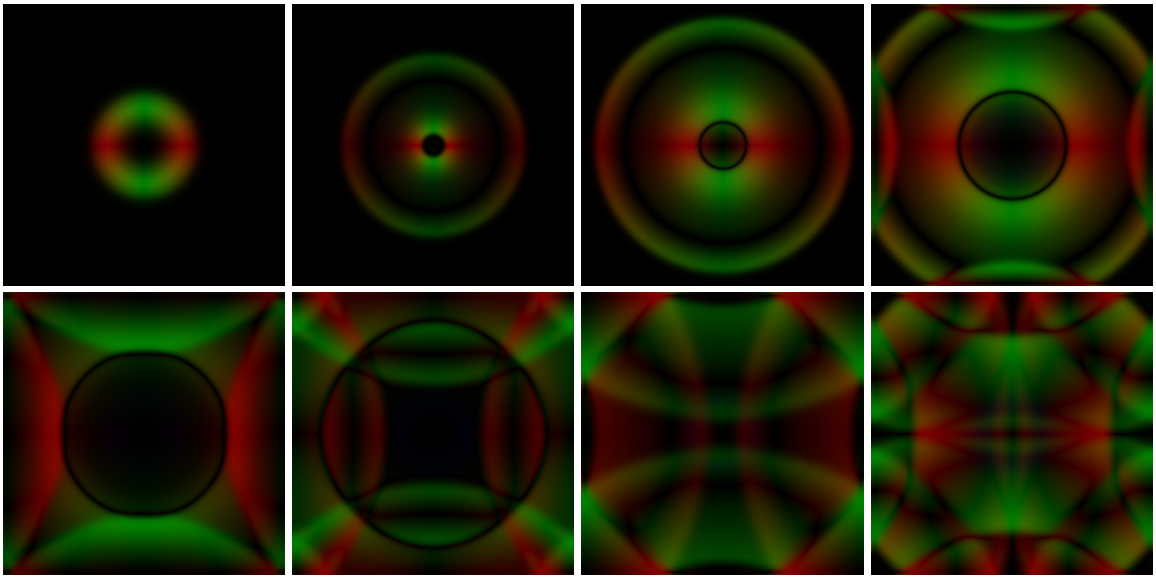


Figure 5.3: Visualization of the velocity field in the *radial explosion* test. The slice was taken in the xy plane from the middle of the computational domain. Absolute velocity in the direction of the x and y axes is colored red and green, respectively. Colors are scaled to the range of the maximum and minimum values in the velocity field.

Boundary conditions were computed at the beginning of each substep, after which the integration kernel was called with the number of the substep as a template parameter. As the α and β coefficients discussed in Section 3.2 depend on the number of the substep, we measured the error after all three substeps had been completed.

Figure 5.3 shows a visualization of the velocity field in the *radial explosion* test. Figures 5.4 and 5.5 show the floating-point error after a full integration step for single and double precision. For reference, we have included also the absolute error for all fields in Figures 5.6 and 5.7.

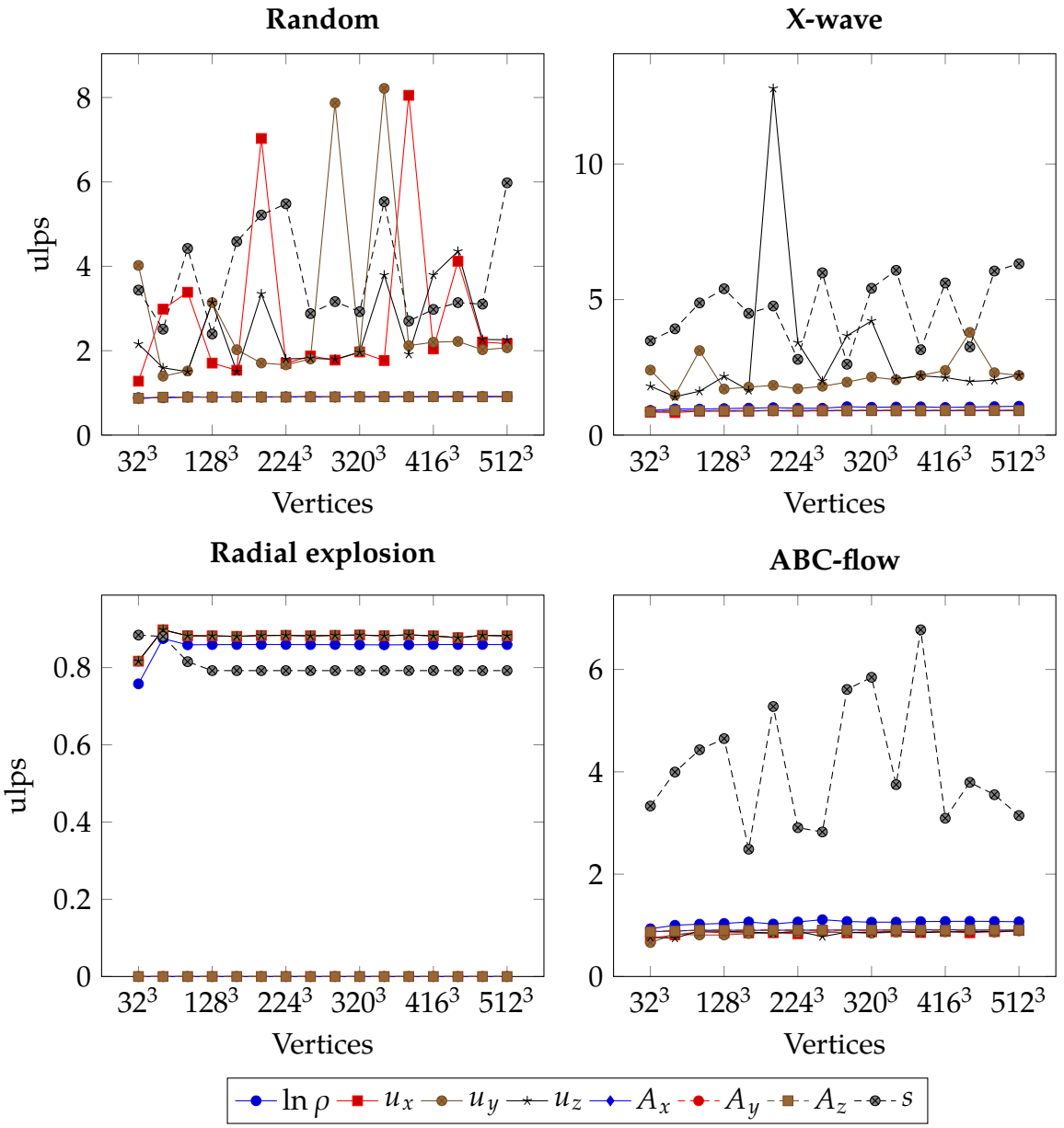


Figure 5.4: The maximum arithmetic error in terms of units in the last place for each field after a complete integration step using 32-bit precision. A single ulp was equal to $\beta^{e-(p-1)}$ as shown in Equation 5.2. The error in ulps at the point of the maximum absolute error of the field is shown.

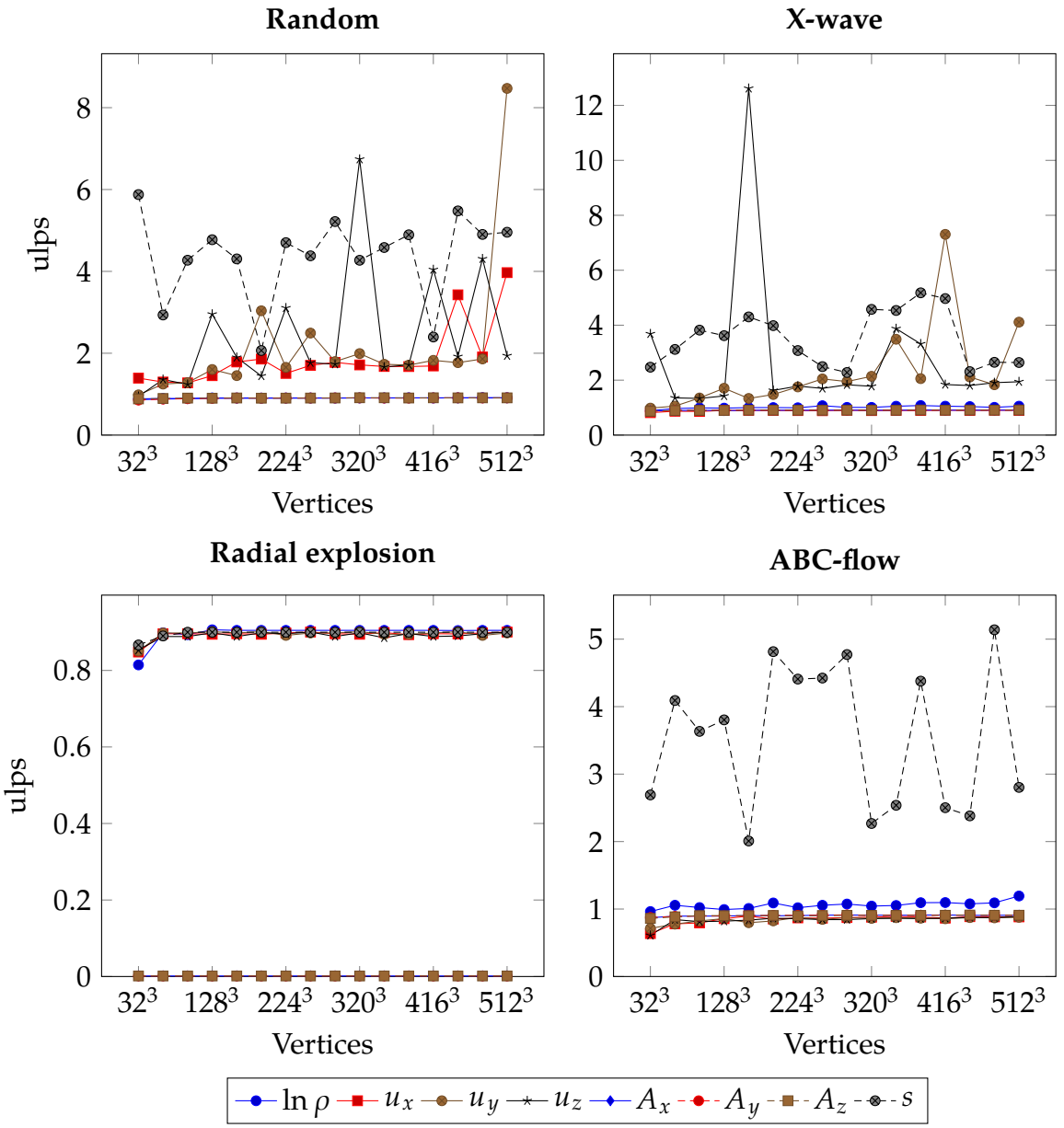


Figure 5.5: The maximum arithmetic error in terms of units in the last place for each field after a complete integration step using 64-bit precision. A single ulp was equal to $\beta^{e-(p-1)}$ as shown in Equation 5.2. The error in ulps at the point of the maximum absolute error of the field is shown.

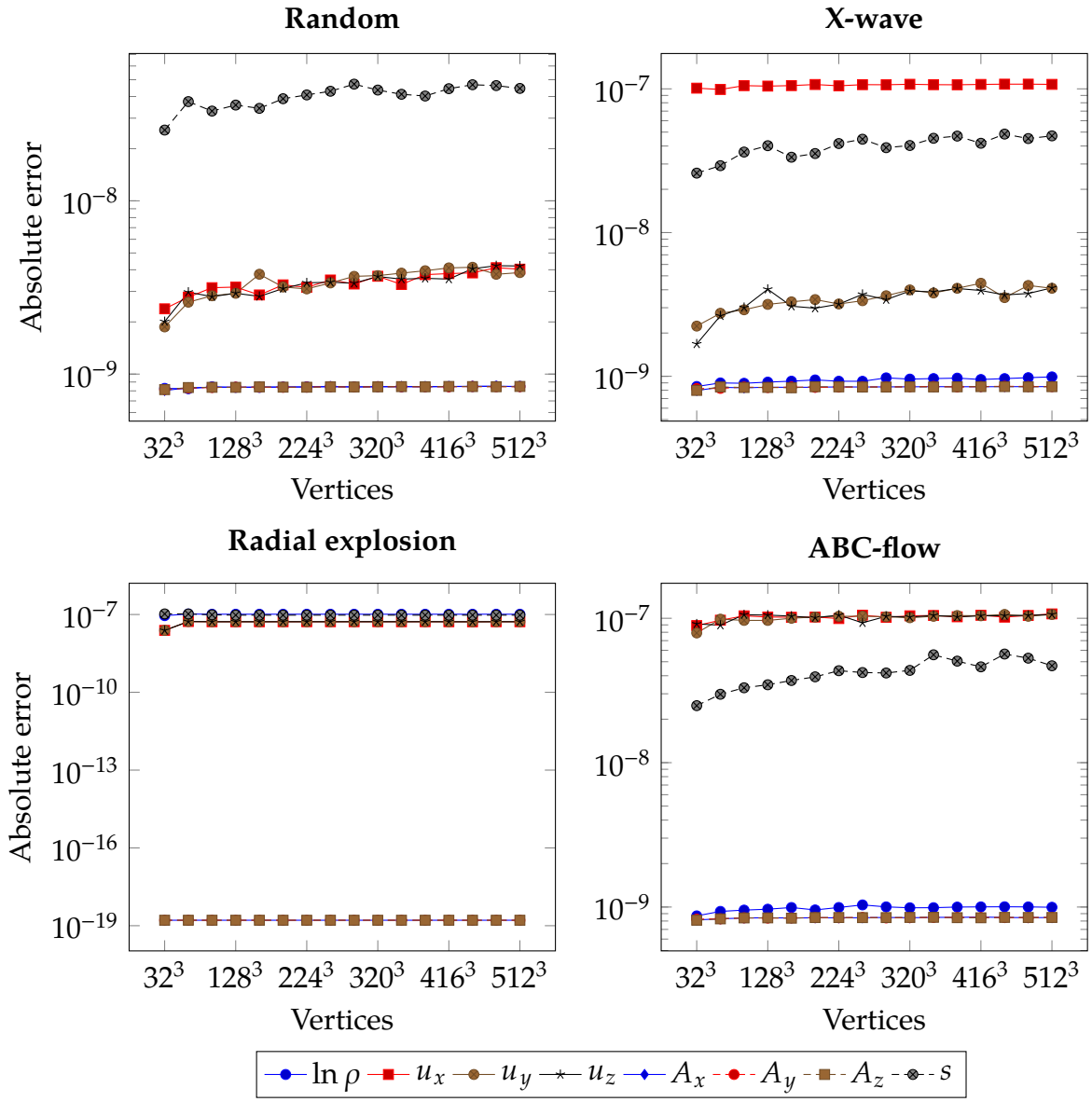


Figure 5.6: The absolute error of an integration step when comparing the output computed with 32-bit precision to a model solution computed with 80-bit precision.

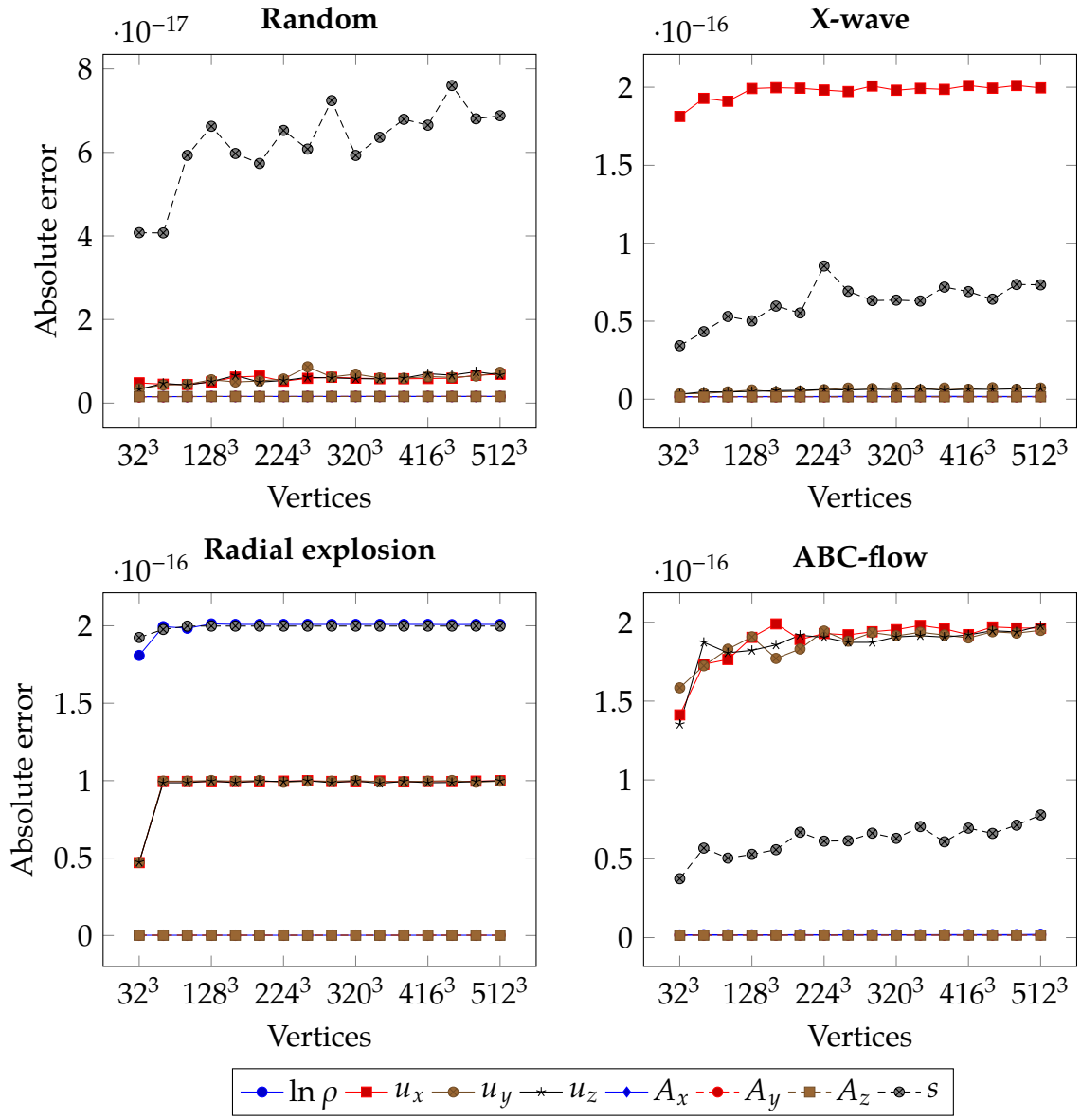


Figure 5.7: The absolute error of an integration step when comparing the output computed with 64-bit precision to a model solution computed with 80-bit precision.

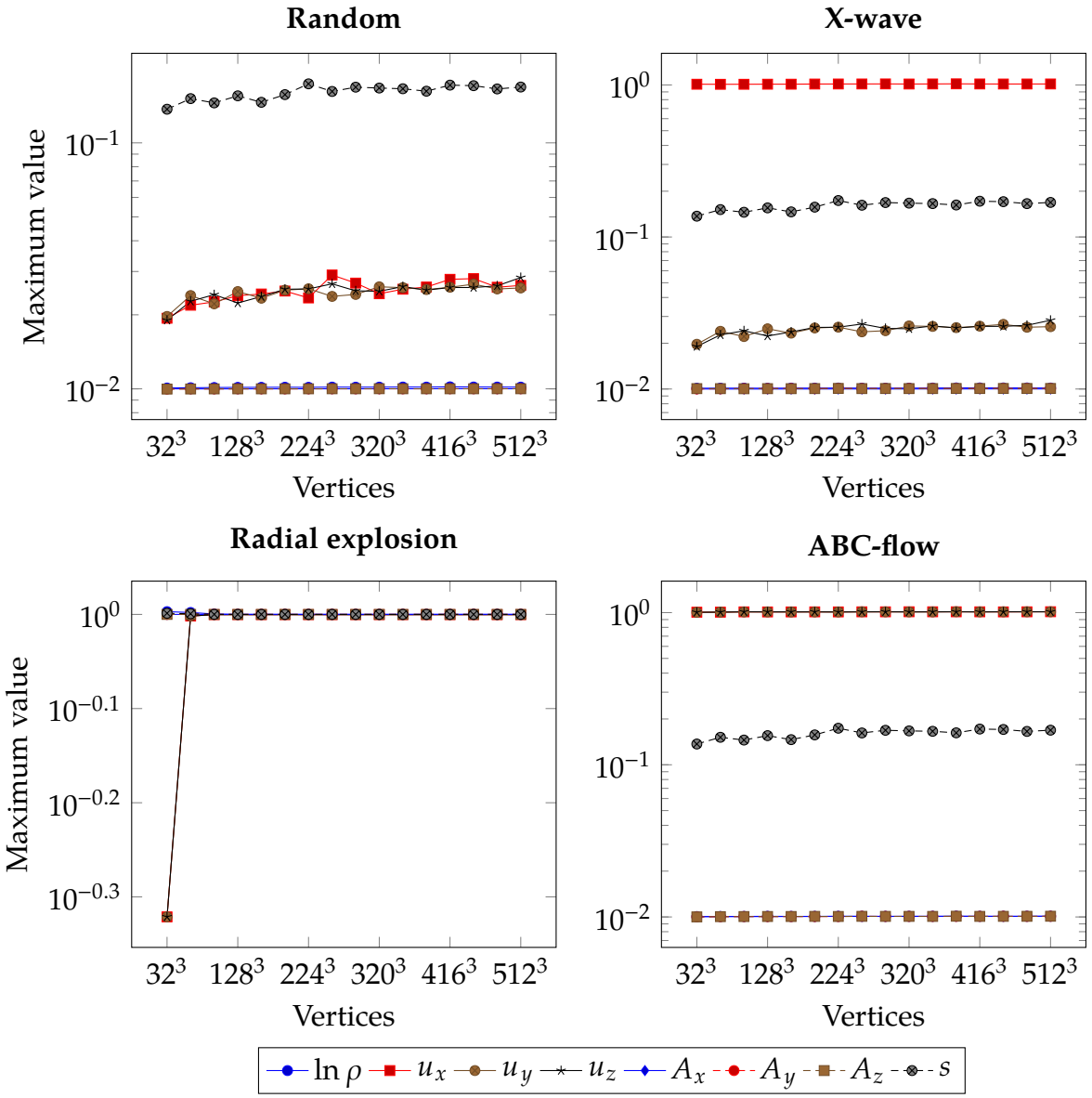


Figure 5.8: Magnitude of the maximum value stored in each field after an integration step using 32-bit precision.

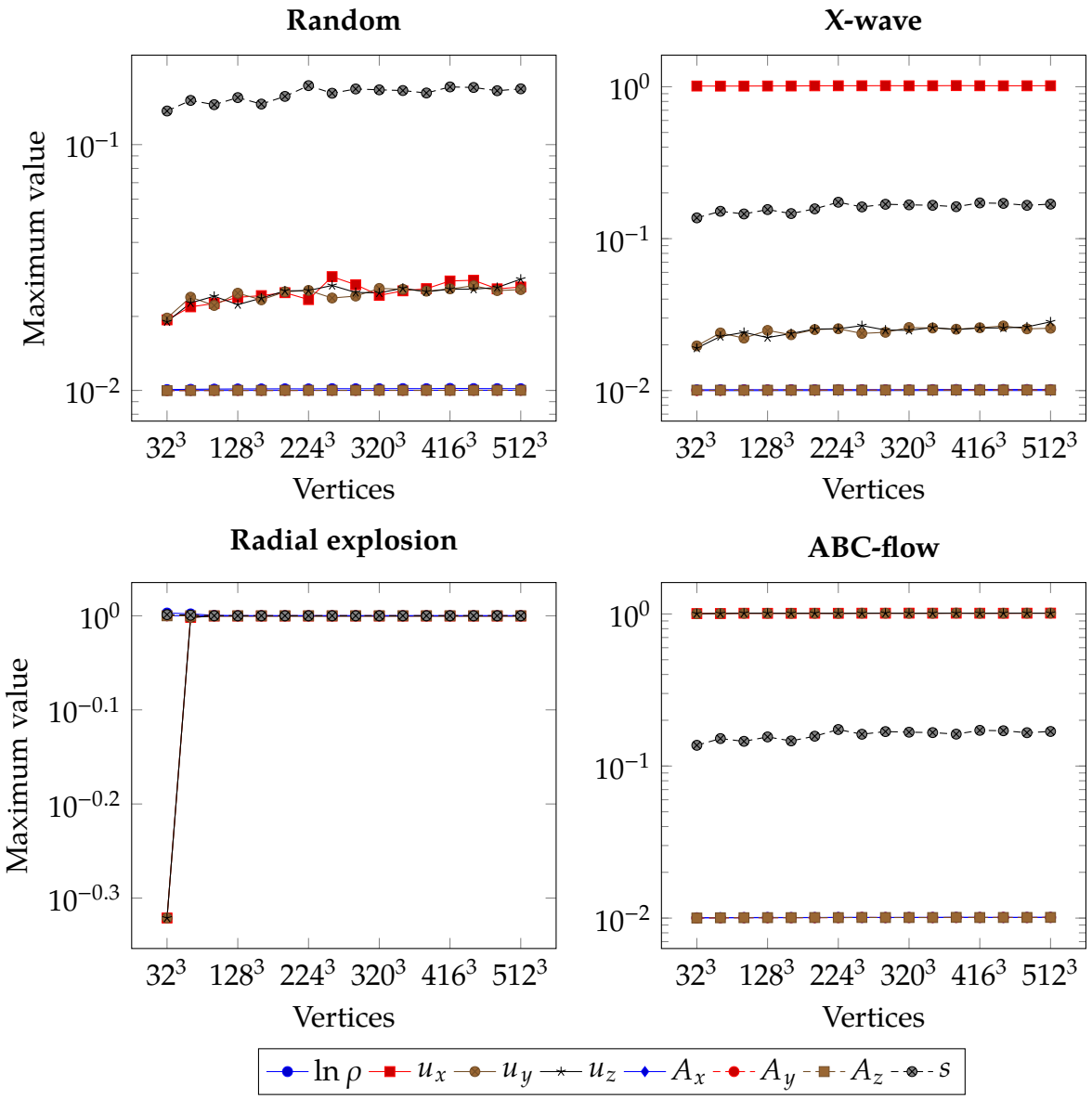


Figure 5.9: Magnitude of the maximum value stored in each field after an integration step using 64-bit precision.

5.2 Hardware utilization

We measured hardware utilization in several benchmarks, where we ran the simulation for 1000 steps and used the running time of the step at the 95th percentile unless otherwise mentioned. We ran the simulation for 10 warm-up steps before measuring the running times to avoid skewing the results due to cold cache misses and the overhead caused by the initialization of the CUDA context. All benchmarks were run on the hardware specified in Chapter 5. While we benchmarked the performance with both single and double precision, we refer to the benchmarks computed with double precision throughout this section unless otherwise stated.

First, we determined the most expensive subroutine when computing an integration step. A breakdown of the relative running times of the subroutines is shown in Figure 5.10. The computation of integration substeps dominated the computation, taking roughly 90% of the running time. The rest of the time was used for computing the boundary conditions. The running times of integration substeps two and three were nearly identical while the initial substep was roughly 10% faster. This was expected, as the state $s - 2$ is not read when solving substep $s = 1$, as shown in Equation 3.25.

Second, we analyzed the performance of computing the final substep as it the most expensive subroutine and, apart from the coefficients α and β used, identical with the computations performed during the second integration substep. Figure 5.11 shows the deviations in the running time of the final substep. Detailed metrics of hardware utilization are shown in Table 5.1.

Third, we measured the operational intensity of the final substep to evaluate whether the performance of the kernel should be expected to be bound by memory bandwidth or compute performance. The measurements are shown in Table 5.2. The operational intensity with single and double precision was 6.9 and 2.7 flops per byte, respectively. The effective operational intensity was likely lower because the fused multiply-add (FMA) operation was counted as two flops in the measurements [104], while the FMA instruction can be issued during a single clock cycle and it has the same latency as, for example, floating-point multiplication [26, 79]. In Section 4.3 we demonstrated that, based on Williams' roofline model [95], the operational intensity required to saturate the arithmetic units of a Tesla P100 PCIe is approximately 13. Because the operational intensity of our kernel was lower, we expected its performance to be bound by memory bandwidth.

As the final test, we compared the running time of the final substep with the

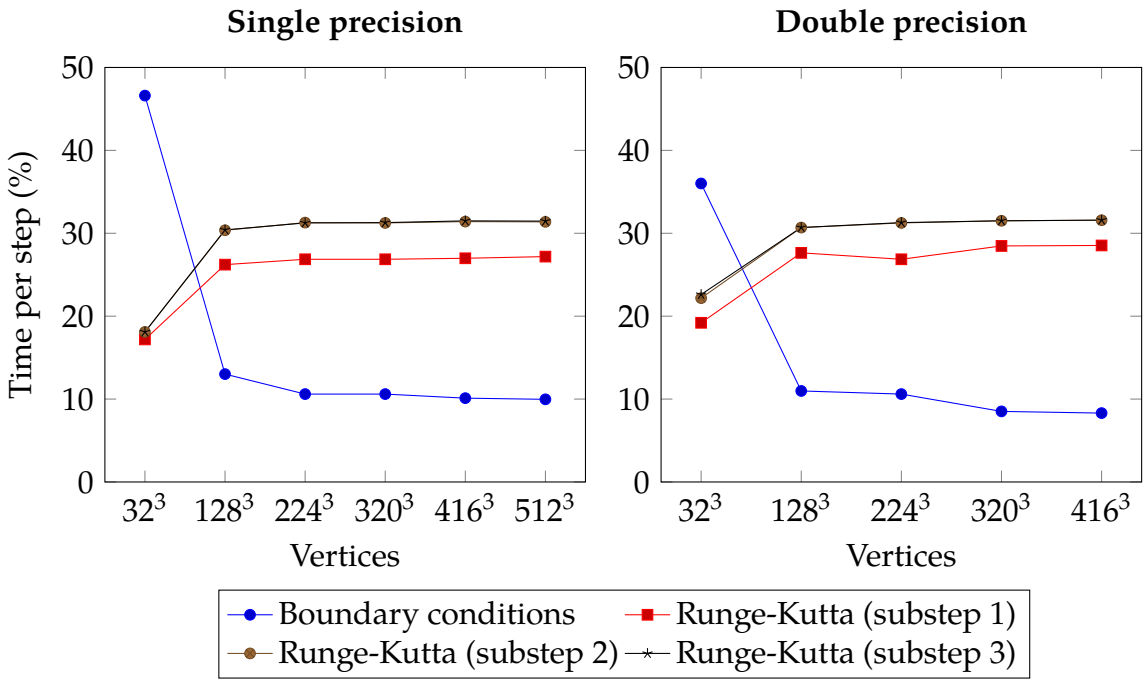


Figure 5.10: Time taken by subroutines used to compute a full integration step. The running times of Runge-Kutta substeps two and three were nearly identical. Integration dominates the computation of a time step, taking roughly 90% of the total running time.

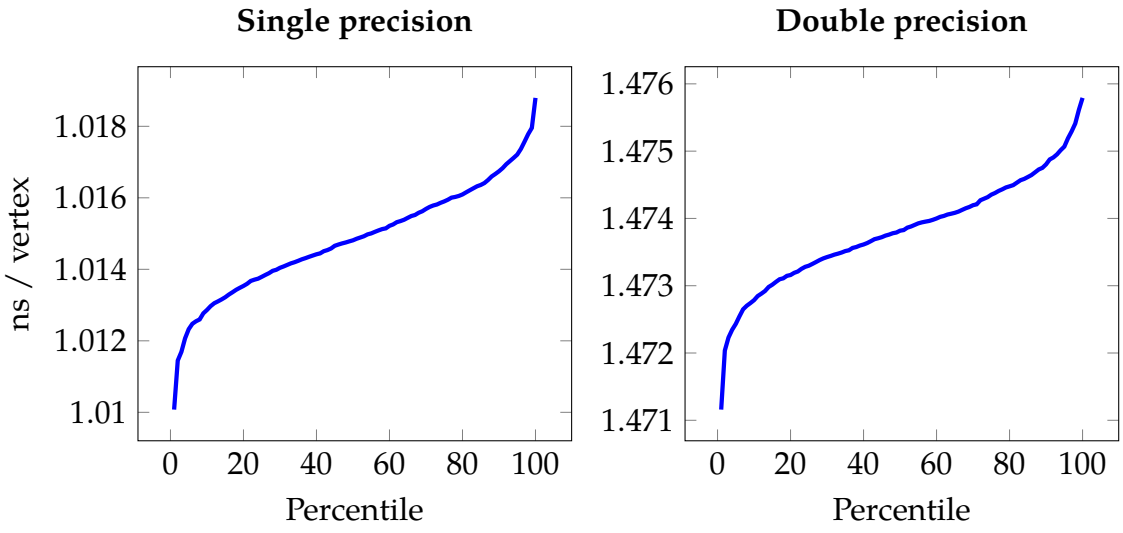


Figure 5.11: Percentiles of the running time of the final integration substep in terms of time used per vertex update when using sixth-order finite differences. The test was conducted with a grid consisting of 256^3 vertices.

Table 5.1: The hardware utilization of a kernel computing the final substep in a grid consisting of 256^3 vertices. Utilization of the single- and double-precision function units (SP/DP FUs) is expressed relative to the peak aritmetic performance. Texture stalls are caused by the high utilization of the texture subsystem or a high number of texture requests. An execution dependency stalls the pipeline when operands required by an instruction are not yet available. With Pascal architectures, a single, unified cache provides functionalities of both the L1 and texture caches [67]. The metrics were measured with NVIDIA's nvvp and nvprof tools provided with the CUDA toolkit version 10.0.130.

Metric	**Single precision**	**Double precision**
Memory bandwidth	130 GiB/s	240 GiB/s
L2 cache bandwidth	700 GiB/s	990 GiB/s
Unified cache bandwidth	2 500 GiB/s	3 700 GiB/s
Registers per thread	255	255
Threads per CTA	128	128
Shared memory usage	0 bytes	0 bytes
L2 hit rate	85%	80%
L1 hit rate	73%	70%
Issue Slot Utilization	27%	22%
Load/Store FU Utilization	10%	10%
SP/DP FU utilization	11%	17%
Occupancy	12%	12%
Primary stall reason	Texture (53%)	Texture (43%)
Secondary stall reason	Execution (25%)	Execution (25%)
Bound by	Latency	Unified cache bandwidth

Table 5.2: Floating-point operations performed and bytes transferred during the final integration substep. The test was conducted using the using NVIDIA's nvprof tool version 10.0.130 and querying metrics `flop_count_sp`, `flop_count_dp`, `dram_read_bytes` and `dram_write_bytes`. The grid consisted of 256^3 vertices. The fused multiply-add operation was counted as two floating-point operations [104]. We do not know why 32-bit floating-point instructions were reported as called when computing the substep in double precision. We strived to ensure that all constants are cast to correct precision before using them in arithmetic operations and that only functions, that return the correct type are used. However, a rigorous examination of the cause is required in future work.

Precision	**Flops (32-bit)**	**Flops (64-bit)**	**Bytes read**	**Bytes written**
Single	$1.80 \cdot 10^{10}$	0	$1.95 \cdot 10^{9}$	$6.60 \cdot 10^{8}$
Double	$6.71 \cdot 10^{7}$	$1.91 \cdot 10^{10}$	$4.41 \cdot 10^{9}$	$2.68 \cdot 10^{9}$

theoretical minimum time, which it would take to read the required data in the simulation domain and write the results back to device memory exactly once. The theoretical minimum running time gave us a conservative bound, which is unattainable in practice because the finite size and bandwidth of caches, and the latency of arithmetic operations are taken into account. However, the bound is useful for determining how efficiently the data is reused and what is the maximum speedup that can be achieved if the algorithm is further optimized.

First, the theoretical maximum bandwidth for double data-rate high-bandwidth memory v2 [81] is calculated by

$$\text{Bandwidth}_{\text{B/s}} = 2 \cdot \text{Bus width}_{\text{b}} \cdot \text{Memory clock frequency}_{\text{Hz}} \, /8 \, . \tag{5.11}$$

For a Tesla P100 PCIe GPU, the memory bus width is 4096 bits and the memory clock frequency 715 Mhz [26, 75]. The three-dimensional computational domain consists of $n_x n_y n_z$ vertices. When including the ghost zones required for computing derivatives near the boundaries and using stencils of order l, a total of

$$(n_x + l)(n_y + l)(n_z + l) \tag{5.12}$$

vertices must be read to calculate the rate of change for a single field. Additionally, when solving integration substeps $s \geq 2$, the values at state $s - 1$ must be read and the result must be written back to device memory as shown in Equation 3.25. Both of these operations access $n_x n_y n_z$ vertices. Therefore if the caches were sufficiently large to hold all data in on-chip memory, the minimum number of reads and writes required with a single field from device memory is

$$\text{Read-writes} = (n_x + l)(n_y + l)(n_z + l) + 2n_x n_y n_z \tag{5.13}$$

vertices. The theoretical minimum time required for reading and writing with w fields can now be computed with

$$\frac{w \cdot \text{Precision}_{\text{B}} \cdot \text{Read-writes}}{\text{Bandwidth}_{\text{B/s}}} \, . \tag{5.14}$$

We used Equation 5.14 to express the efficiency of our integration kernel as the ratio of the theoretical minimum running time to the measured running time of our kernel. The comparison of our solver against the theoretical minimum running time is shown in Figure 5.12 when using stencils visualized in Figure 5.13.

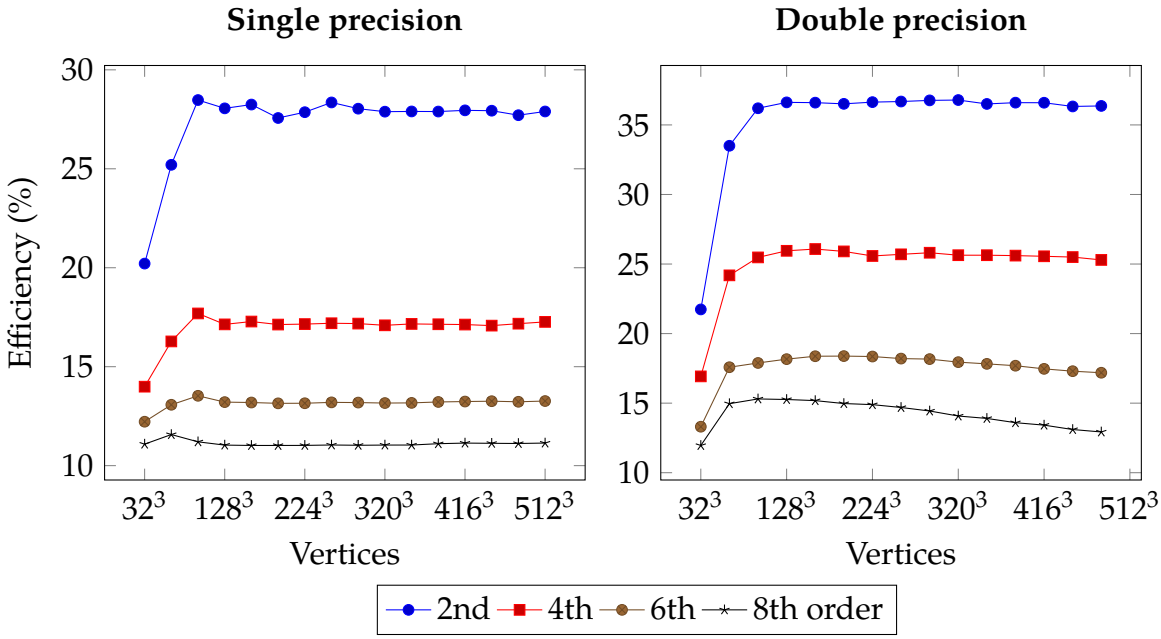


Figure 5.12: Efficiency of computing the final integration substep as a ratio of the theoretical minimum running time to the measured running time at the 95th percentile when using second-, fourth-, sixth-, and eight-order finite differences. We decomposed the grid to blocks of $(32, 1, 4)$ vertices, where each block was operated by a cooperative thread array (CTA). The dimensions of the CTA were selected by auto-tuning the code to achieve maximum performance with double precision in a grid consisting of 256^3 vertices. The maximum number of vertices that could be stored in the device memory of a Tesla P100 PCIe was 480^3. The stencils used in this test are visualized in Figure 5.13.

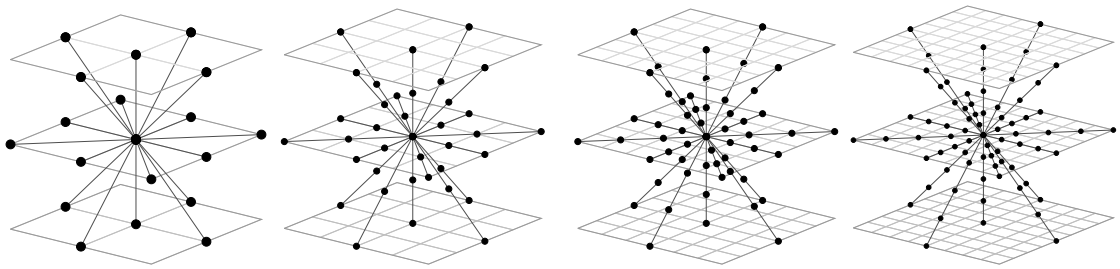


Figure 5.13: Visualization of the stencils used in this work. The second-, fourth-, sixth-, and eight-order stencils are listed from left to right.

5.3 Comparison with a CPU solver

As the final test, we compared our solver with the Pencil Code [19]. We chose the Pencil Code because it has a mature code base, it has been used for generating data for numerous publications, and we are planning to interface our GPU solver with it in the future. The performance comparison of our solver and the Pencil Code is shown in in Figure 5.14. The test case used with the Pencil Code is available at [88].

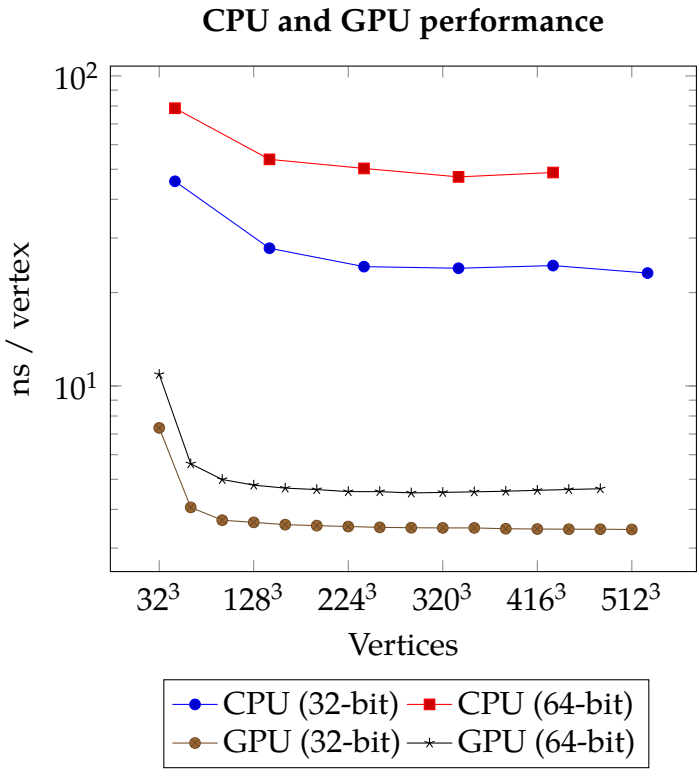


Figure 5.14: The performance comparison of our solver and the Pencil Code [19]. A logarithmic scale was used for the y axis. Our solver was run on a Tesla P100 PCIe GPU, while the Pencil Code was run on a total of 24 cores on two Intel Xeon E5-2690 v3 processors within a single compute node using Intel MPI version 14.0.1. Both solvers computed the equations defined in Chapter 3 and used a constant time step δ = 1e−4. The optimal grid dimensions for the CPU solver were multiples of the available core count, in our case 24, while for the GPU solver the optimal dimensions were multiples of the warp size, 32. The largest problem size that could fit into the memory of a Tesla P100 PCIe GPU when using double precision was 480^3 vertices. The CPU solver returned an unknown error when attempting to run the test with double precision and a grid consisting of 528^3 vertices. The best performance of the GPU solver was 3.5 and 4.6 ns / vertex / full integration step for single and double precision, respectively. The best performance with the CPU solver was 23 and 47 ns / vertex / integration step.

Chapter 6

Discussion

In this work, we created a library and a domain-specific language for stencil computations on GPUs. In our test case, we simulated the flow of compressible, electrically conducting fluids with second-, fourth-, sixth-, and eight-order finite differences and advanced the simulation using the third-order Runge-Kutta method. The performance of the integration kernels generated with our compiler was within an order of magnitude of the theoretical maximum limit, achievable by executing a perfect algorithm on a machine providing infinitely large and fast caches, and latency-free arithmetic. To our knowledge, this is the first work where the performance bounds of computations with complex, high-order stencils have been analyzed in detail, and reported that a performance of 18% of a conservative hardware limit has been achieved in simulations of magnetohydrodynamics with sixth-order finite differences.

Our solution relied on three techniques to provide efficient kernels for computations with arbitrary stencils on various architectures. First, the integration kernel was generated in a way, which decoupled reading, computation and writing to stages, where the output of one stage was passed to the next. The benefit of this approach was, that expensive reading operations could be preprocessed, and the result could be stored into local memory and reused in further computations. Second, we relied on implicit caching to reduce traffic to off-chip memory, which had the benefit of not having to manually tune the implementation for stencils of different orders, and not requiring synchronization between CTAs. Because we allocated a significant amount of resources per CTA to improve data reuse with the cost of having fewer warps multithreaded on the SIMT processors of a GPU, explicit synchronization would have likely stalled the execution. Finally, we used auto-tuning to find the optimal problem decomposition for any given architecture.

We verified our results by making a comparison between a model solution solved

with 80-bit precision and our GPU solver, which used 32- or 64-bit precision. When computing the boundary conditions and scalar reductions, our GPU results agreed exactly with the model solution because the computations involved only copying and logical operations, and no arithmetic operations were performed with the data. With vector reductions, the error ranged from 0 to 0.8 units in the last place (ulps). While a single arithmetic operation with floating-point numbers conforming to the IEEE 754-2008 standard should yield an absolute error less or equal to $\frac{1}{2}$ ulps, as stated in Equation 5.4, computing the length of a vector requires 7 arithmetic operations if fused multiply-add is not used. Therefore it is reasonable to expect that the error propagates when arithmetic is performed with rounded numbers.

The error accumulated with our integration kernel was notably higher, at most 8.5, 12.7, 1.0, and 5.2 ulps, for *random*, *X-wave*, *radial explosion* and *ABC-flow* test cases, respectively. We noted that the maximum arithmetic error depended significantly on the chosen initial values and configuration parameters. However, as the CPU model solution was computed with exactly the same functions as the GPU solution, but in higher precision, and the error does not vary between runs, we do not suspect there are issues with the parallelization of the algorithm.

We made an additional sequential test solely on a CPU, where we compared the error of terms $1/\rho$, $1/T$, and $1/(\rho T)$ computed with 32 and 80 bits of precision. We selected these terms because $(\rho T)^{-1}$ was used to solve Equation 3.12 and solving it required using the exponential function, which potentially yields errors larger than $\frac{1}{2}$ ulps because it is not specified by the IEEE 754-2008 standard [68]. We performed the test by computing

$$\frac{1}{\rho} = e^{-\ln \rho} \tag{6.1}$$

and

$$\frac{1}{T} = e^{-\ln T} = e^{-[\gamma s / c_p + (\gamma - 1) \ln \rho]}, \tag{6.2}$$

where we used $\gamma = c_p = 2$ and computed the error with different magnitudes of s and $\ln \rho$. We made the observation that the results of $1/\rho$ and $1/(\ln T)$ were highly inaccurate, 100 ulps or larger, if either $\ln \rho$ or s was of magnitude 10^1 or higher. When $|\ln \rho| \leq 1$ and $|s| \leq 1$, the maximum errors were 1.17, 3.38 and 15.87 ulps for $1/\rho$, $1/\ln T$ and $1/(\rho T)$, respectively. The source code of this test is available at [88]. This test supports the notion, that performing arithmetic operations with inaccurate intermediate results contributed to the errors measured in our verification tests.

We noted that if magnitudes of the input values varied widely, significant digits of the smallest values would be likely lost due to cancellation. These types of

cancellation errors could potentially be mitigated by reformulating some of the equations [99]. In Figures 5.6 and 5.7, we see that the magnitude of the absolute error scales with the precision used when contrasted with the maximum values used in the computations, shown in Figures 5.8 and 5.9. We observed that the velocity field propagates symmetrically in the *radial explosion* test case shown in Figure 5.3, which suggests that the boundary conditions and the computations with the velocity field are performed according to the specification.

We also noted that NVIDIA's profiling tool, nvprof, reported single-precision operations being performed when solving the integration substep with double precision, as shown in Table 5.2. While our review of the code suggests that all arithmetic operations are performed with correct precision, further research is needed to determine the cause. If some of the operations were performed in single precision, then this would likely have a degrading effect on accuracy. However, the effect on performance would likely be minor, as the proportion of single-precision instructions to the total number of floating-point instructions performed in the kernel is only 0.35%.

Because updating each vertex involves a large number of arithmetic operations with rounded intermediate results, our experiments suggest that intermediate computations may yield errors that are at least as high as 15.87 ulps, the measured absolute error scales with the precision used, and the errors do not vary between runs, we conclude that the errors are likely caused by performing the arithmetic operations with finite precision. Because we do numerous calculations in the kernel, it is not feasible within the bounds of this work to formally prove that the arithmetic error is within acceptable bounds, for example by determining the condition number for the equations. The condition number can be used to approximate how much the relative rounding error is magnified when some function is evaluated [100].

We compared the performance of our solver with the Pencil Code [19], which is a mature project focusing on simulating astrophysical magnetohydrodynamics (MHD) and turbulence efficiently with high-performance computers. With both solvers, we simulated MHD by solving the equations described in Chapter 3. The exact test case used with the Pencil Code is available at [88]. Our solver outperformed the Pencil Code, executed on a total of 24 CPU cores via MPI, by factors of 6.7 and 10.4 with single and double precision, respectively. Given the thermal design power of 300 Watts of a Tesla P100 PCIe [8] and 135 Watts of a single Intel Xeon E5-2690 v3 processor [7], not including the power requirements of other components in the server blade, the performance per Watt was 6.0 and 9.4 better with a Tesla P100 PCIe with single and double precision, respectively. This suggests that GPUs can be utilized efficiently for solving problems, which involve computations with a large

number of coupled fields using high-order finite differences.

While we strived to make the comparison of CPU and GPU solvers as fair as possible, making a direct comparison is difficult because of differences in architectures, the execution model and connectivity between processors. A single high-performance computing node houses typically more GPUs than CPUs, and in our test we utilized only one of the available four Tesla P100 GPUs on the node. While the main memory is shared among CPUs on multi-socket motherboards, communication between GPUs must be done via either the PCIe bus or a specialized interconnect. This interconnect, NVLink, provides 19 GiB/s bandwidth per direction between pairs of GPUs [8, 26, 75] while PCIe 3.0 x16 provides a directional bandwidth of 15 GiB / s [75]. While our results suggest that a single Tesla P100 GPU outperforms two Intel Xeon E5-2690 v3 processors in raw performance, our tests did not account for the overhead caused by inter-GPU communication if the library is extended to employ multiple GPUs. However, our preliminary tests within a single node suggest that inter-GPU communication can be mostly hidden by updating a subset of the computational domain simultaneously with communicating the ghost zones. In our preliminary tests, we noted that the performance may scale to multiple GPUs within a node with roughly 90–95% efficiency.

The size of the code required for expressing kernels with our domain-specific language (DSL) was notably smaller than was required for expressing highly-optimized CUDA kernels. By focusing on a narrower problem domain, we were able to simplify the syntax of our DSL to include only features, which were necessary to translate the code to efficient CUDA kernels. In this paragraph, we use the term *word* to refer to a non-zero-length string delimited by whitespace. The solver used to compute the full set of MHD equations, shown in Appendix C, could be written in 558 words. The solver was compiled into 1098 words of CUDA code, which was embedded in a header consisting of the built-in functions described in Section 4.2, and the necessary code for calling the kernel from host code. The complete header consisted of 3826 words in total.

The hardware was utilized most efficiently when computing derivatives with second-order stencils with double precision. This was expected, as the working set required for processing small stencils was likely to fit in on-chip caches. In contrast, the working set for processing larger stencils was more likely to spill from registers and L1 to slower memory before the data had completely been exhausted. As we did not account for the size or bandwidth of the cache when computing the theoretical minimum running time, we saw a systematic reduction in efficiency when increasing the stencil size. Surprisingly, computations with double precision were relatively

more efficient than with single precision. After analyzing the disassembly of the source code, we noted that memory transactions to and from global memory were done with 64-bit-wide loads and stores. With single precision, data from device memory were fetched with 32-bit instructions. Therefore with double precision, the memory bus was utilized more efficiently as fewer transactions were needed to transfer the same amount of data. In addition, 32-bit integer and 64-bit floating-point instructions are executed on different functional units on the Pascal microarchitecture [8] and the warp schedulers can dual-issue independent instructions from a warp to different functional units during the same clock cycle [67]. Therefore, to our knowledge, instructions used to calculate indices could be issued during the same clock cycle as the 64-bit floating-point operations used in processing the stencils, which potentially improved instruction-level parallelism in the kernel.

With double precision, instruction latencies could be hidden because of the more efficient utilization of the memory bus and improved instruction-level parallelism, and the performance was bound by the unified cache bandwidth. With single precision, not enough instructions could be generated to saturate the memory and arithmetic systems with work, therefore latencies could not be hidden and the performance was bound by instruction latency, as shown in Table 5.1. Modern GPUs support vectorized memory access for up to 128-bit wide words [9, 34], which offer better efficiency in terms of instructions to bytes transferred but, depending on the problem, may also increase register pressure [65]. The performance for computations with single precision could potentially be improved by packing the data in `float2` and `float4` structures, which are serviced with vectorized loads and stores.

The primary stall reason for both single and double precision was texture fetching. This was expected, as we declared the input arrays with the `const __restrict__` type qualifier, which indicates to the compiler that input arrays could be read through the texture cache. By inspecting the assembly code, we confirmed that the CUDA compiler did generate instructions to read the input arrays through the texture cache. With double precision, the performance of integration kernels was limited by the bandwidth of the unified cache.

The secondary stall reason was execution dependency, where the operands of an instruction are not yet available. As discussed in Section 2.1, latencies caused by stalls, such as execution dependencies, can be mitigated by increasing the number of warps being multithreaded on the SIMT processors or ensuring that there are enough independent instructions in-flight [65].

In Figure 5.12 we saw a slight downward trend in efficiency when the problem size was increased. The downward trend was more pronounced when using higher-

order finite differences. The difference between the highest and lowest efficiency when processing eight-order stencils was 2.4 percentage units for grids containing 96^3 or more vertices. We suspect that the downward trend is partially caused by an increase in translation-lookaside buffer (TLB) misses. Because we store arrays linearly in memory, the physical addresses of data stored in neighboring vertices may be far apart. With the library, we map a vertex at (i, j, k) belonging to the field w to a one-dimensional index at $i + jn_x + kn_xn_y + wn_xn_yn_z$, where (n_x, n_y, n_z) are the dimensions of the grid. Jia *et al.* [26] state that the coverage of L1 and L2 TLB of a Tesla P100 GPU is 32 MiB and 2048 MiB, respectively. With an order l difference scheme, cache size c and precision p in bytes, if there are m fields where each field is allocated one input and output array, both containing $(n+l)^3$ vertices, where $n = n_x = n_y = n_z$, we would expect to see a decline in performance when $n > \sqrt[3]{c/(2mp)} - l$. With a Tesla P100 PCIe, the n required to match the coverage of L1 and L2 caches would then be $n = 56$ and $n = 248$, respectively, when using eight-order differences, double precision and updating eight fields. However, in Figure 5.12, we see a drop when the computational domain of a grid contains more than 96^3 vertices. Therefore L1 and L2 TLB misses can not solely explain the downward trend. To our knowledge, TLB misses can not be measured directly with the available profiling tools. We made a non-exhaustive experiment by using shared memory instead of implicit caching, but this test also suggested a similar downward trend. We also tried invoking the integration kernel several times, where each kernel operated on a subset of the computational domain. With this approach, we were able to increase efficiency at larger grid sizes, which suggests that performance loss of solving larger grids could be mitigated by solving the problem with multiple kernels, which access data from fewer addresses. If the downward trend were caused by TLB misses, performance could potentially also be improved by changing from linear indexing to a mapping which preserves spatial locality, such as Morton order, and storing the data as an array of structures.

However, we expect that communicating the ghost zones will be a bottleneck when the library is extended to work on multiple nodes. Because communication across nodes is at least an order of magnitude slower than communication within a node, we expect that the data must be packed in a way, such that it can be communicated with as few transactions as possible while simultaneously performing computations with a subset of data local to the node. Therefore we suspect that the optimal layout for the data when using multiple nodes is different from what is optimal for performing the computations on a single GPU. Additionally, the performance loss of the suspected TLB misses was not significant enough to warrant

deeper investigation, which is why we have left exploring different data layouts to future work.

As a final remark on hardware utilization, we note that the theoretical maximum performance used in the efficiency comparison of our integration kernel is highly conservative, and not attainable in practice. In real applications, the cache size, latency and bandwidth are limited, and common floating-point arithmetic operations take 6–14 cycles to complete [26], which was not taken into account when calculating the theoretical maximum performance. On Pascal architectures, integer addition and multiplication are emulated, and have been reported to require 86 cycles to complete [26]. Additionally, the effective bandwidth from device memory with ECC is roughly 70–85% of the theoretical maximum bandwidth [26, 34, 105]. Therefore, while our implementation is at most 9.1 times slower than the conservative roofline performance, any improvements over our algorithm would likely show more modest speedups in practice.

Ensuring that a sufficiently large number of warps are multithreaded on the SIMT processors is often recommended as a straightforward optimization technique for GPU programs [9]. However, we argue that striving to minimize the number of device memory transactions by aggressively utilizing caches, which may come at the cost of lower occupancy as suggested by Volkov [62], is necessary to attain better performance in memory-bound problems. As the gap between operational performance in relation to the number of bytes that can be transferred with the memory system per second continues to increase [23, 74], more problems will become bound by the memory bandwidth with future architectures. Improving reuse and increasing occupancy are often mutually exclusive, because allocating more resources per thread reduces the total number of warps that can be multithreaded on a SIMT processor. This raises the issue of how the latency of memory and other operations can be hidden if there are only a few threads, which are all stalled. Instruction-level parallelism (ILP) has been discussed widely in literature [2, 3, 64, 95] and was reported by Volkov [62, 65] to be effective in hiding latencies even at low occupancies, which we also observed in this work.

In future work, we will extend the library to scale to multiple GPUs within and across nodes. With multiple nodes, we will evaluate whether it is beneficial to utilize existing MPI solutions, such as the one provided with the Pencil Code, or whether to develop a standalone module that is optimized for communicating with GPUs across a network. As for the compiler, we will extend it to generate vectorized load instructions and evaluate, whether it would be beneficial to utilize optimizing compiler infrastructures, such as LLVM [106]. Current GPUs support

vectorized loads and stores from device memory, while arithmetic is performed generally on scalar stream processors [9, 25, 43, 55, 70]. Further research is needed to determine whether we would see performance improvements in using other compiler frameworks optimized for parallel computing, such as Delite [18] and Lift [17].

As the majority of supercomputers utilize NVIDIA GPUs, we implemented the library in CUDA in this work. Our final point of interest in future work is to evaluate, whether it would be useful to port the library to a platform-independent API, such as OpenCL [10] without suffering a significant penalty in performance. However, AMD is working on a project called heterogenous-computing interface for portability (HIP)[1], which is stated to translate CUDA code into a platform-independent language, in which case our library would not have to be modified to work on GPUs of either vendor.

From the aspect of computational sciences, we will add more equations and fields to the test case presented in this work. We have started to use the library in the research of astrophysical turbulence, and will test the library extensively in physical cases in future work. Our preliminary tests suggest that the computation time increases close to linearly, roughly with 90–95% efficiency, with the number of data transfers required when adding more computations with additional fields. Our preliminary tests also indicate that computing derivatives with different stencils than presented in this work is efficient as long as the data access pattern is relatively local. For example, in our preliminary tests we saw no significant difference in execution time when solving mixed derivatives using the direct formulation from the definition of first-order derivatives, in which case the stencil consists of subsets of xy-, xz-, and yz-planes totaling 127 stencil points when using sixth-order finite differences. While first surprising, this could be explained by the fact that as cache lines are replaced in segments of 128 bytes, the number of cache lines that have to be replaced when using either the 55-point tridiagonal stencil or the naïve 127-point stencil is not significantly different.

[1] `https://github.com/ROCm-Developer-Tools/HIP`.

Conclusion

Our research problem was to create a domain-specific language (DSL), which can be used to simulate magnetohydrodynamics and express computations with stencils of various orders, and which can be generated into kernels that achieve a performance within an order of magnitude of the hardware limits.

We reported that high-order stencil computations in a test case encompassing a wide range of physical simulations can be implemented efficiently on GPUs. We created a library and a domain-specific language for expressing stencil computations with a high-level representation of the problem, and demonstrated that a performance within an order of magnitude of a conservative hardware limit can be achieved when the library is used to perform operations with stencils of orders two, four, six, and eight.

We argued that in problems bound by memory bandwidth, it is necessary to find ways to improve reuse, potentially with the cost of occupancy, in order to improve performance further, which has also been suggested in previous work [38, 62]. While we expect that our implementation could be outperformed with an algorithm, which would utilize the caches more efficiently, in any case the maximum speedup over our algorithm would be of a factor of six or less with double precision when using sixth-order finite differences.

We verified our results against a model solution and deemed that the errors were within expectable bounds, and were likely caused by performing several arithmetic operations with finite precision. Finally, we contrasted the performance of our library with an MHD solver widely used for high-performance computing, and reported that with a GPU, we can achieve roughly up to an order of magnitude better performance and power efficiency than with CPUs in high-order stencil computations.

As indicated by previous work [18, 57], we also report that efficient code can be generated from a high-level language constrained to a well-specified problem domain, and that rigorous hand-tuning, requiring expertise in parallel hardware architectures, is not always necessary to obtain competitive performance.

Bibliography

[1] D. Post, "The future of computing performance," *Computing in Science & Engineering*, vol. 13, pp. 4–5, July 2011.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Burlington, MA, USA: Morgan Kaufmann Publishers, 5th ed., 2011.

[3] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Burlington, MA, USA: Morgan Kaufmann Publishers, 5th ed., 2013.

[4] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Communications of the ACM*, vol. 52, pp. 56–67, Oct. 2009.

[5] H. Sutter and J. Larus, "Software and the concurrency revolution," *ACM Queue*, vol. 3, pp. 54–62, Sept. 2005.

[6] N. R. Council, *The Future of Computing Performance: Game Over or Next Level?* Washington, D.C., USA: The National Academies Press, 1st ed., 2011.

[7] Intel, Santa Clara, CA, USA, *Intel Xeon Processor E5-2690 v3*, 2018. [Online]. Available: `https://ark.intel.com/products/81713/Intel-Xeon-Processor-E5-2690-v3-30M-Cache-2-60-GHz-`. [Accessed May 9, 2019].

[8] NVIDIA, Santa Clara, CA, USA, *NVIDIA Tesla P100*, 2018. [Online]. Available: `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`. [Accessed May 9, 2019].

[9] NVIDIA, Santa Clara, CA, USA, *CUDA C Programming Guide*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. [Accessed May 9, 2019].

[10] Khronos OpenCL Working Group, Beaverton, OR, USA, *The OpenCL Specification*, Feb. 2019. [Online]. Available: `https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf`. [Accessed May 9, 2019].

[11] A. Brandenburg, "Computational aspects of astrophysical MHD and turbulence," *Advances in Nonlinear Dynamos*, vol. 9, pp. 269–344, Apr. 2003.

[12] Käpylä, M. J., Käpylä, P. J., Olspert, N., Brandenburg, A., Warnecke, J., Karak, B. B., and Pelt, J., "Multiple dynamo modes as a mechanism for long-term solar activity variations," *A&A*, vol. 589, p. A56, May 2016.

[13] D. E. Keyes, L. C. Mcinnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, X. Jiao, K. Jordan, D. Kaushik, E. Kaxiras, A. Koniges, K. Lee, A. Lott, Q. Lu, J. Magerlein, R. Maxwell, M. Mccourt, M. Mehl, R. Pawlowski, A. P. Randles, D. Reynolds, B. Rivière, U. Rüde, T. Scheibe, J. Shadid, B. Sheehan, M. Shephard, A. Siegel, B. Smith, X. Tang, C. Wilson, and B. Wohlmuth, "Multiphysics simulations: Challenges and opportunities," *International Journal of High Performance Computing Applications*, vol. 27, pp. 4–83, Feb. 2013.

[14] R. Khoury and D. W. Harder, *Numerical Methods and Modelling for Engineering*. New York, NY, USA: Springer International Publishing, 1st ed., 2016.

[15] G. Roth, J. Mellor-Crummey, K. Kennedy, and R. G. Brickner, "Compiling stencils in high performance Fortran," in *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, SC '97, (New York, NY, USA), pp. 1–20, ACM, Nov. 1997.

[16] K. Asanovic, R. Bodik, B. Catanzaro, J. James Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The landscape of parallel computing research: A view from berkeley," tech. rep., EECS Department, University of California, Berkeley, Dec. 2006.

[17] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with Lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, (New York, NY, USA), pp. 100–112, ACM, Feb. 2018.

[18] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Transactions on Embedded Computing Systems*, vol. 13, pp. 134:1–134:25, Apr. 2014.

[19] Nordic Institute for Theoretical Physics, Stockholm, Sweden, *The Pencil Code: A High-Order MPI code for MHD Turbulence. User's and Reference Manual*, July 2018. [Online]. Available: `http://pencil-code.nordita.org/doc/manual.pdf`. [Accessed May 9, 2019].

[20] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic optimization for image processing pipelines," *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 429–443, Mar. 2015.

[21] J. Ragan-Kelley, *Decoupling Algorithms from the Organization of Computation for High Performance Image Processing*. PhD thesis, Massachusetts Institute of Technology, MA, USA, 2014.

[22] J. Stam, "Stable fluids," in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, (New York, NY, USA), pp. 121–128, ACM Press/Addison-Wesley Publishing Co., July 1999.

[23] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 20–24, Mar. 1995.

[24] Intel, Santa Clara, CA, USA, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Apr. 2019. [Online]. Available: `https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf`. [Accessed May 9, 2019].

[25] J. Nickolls and D. Kirk, "Appendix C: Graphics and Computing GPUs," in *Computer Organization and Design: The Hardware/Software Interface*, Burlington, MA, USA: Morgan Kaufmann Publishers, 5th ed., 2013.

[26] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU architecture via microbenchmarking," tech. rep., Citadel Enterprise Americas LLC, Apr. 2018.

[27] D. Blythe, "The Direct3D 10 system," *ACM Transactions on Graphics*, vol. 25, pp. 724–734, July 2006.

[28] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, pp. 56–69, Mar. 2010.

[29] OpenACC-Standard.org, San Francisco, CA, USA, *The OpenACC Application Programming Interface Version 2.7*, Nov. 2018. [Online]. Available: `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf`. [Accessed May 9, 2019].

[30] Apple Inc., Cupertino, CA, USA, *Metal Shading Language Specification, Version 2.1*, Mar. 2019. [Online]. Available: `https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf`. [Accessed May 9, 2019].

[31] The Khronos Group Inc., Beaverton, OR, USA, *The OpenGLGraphics System: A Specification, Version 4.6*, Feb. 2019. [Online]. Available: `https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf`. [Accessed May 9, 2019].

[32] The Khronos Vulkan Working Group, Beaverton, OR, USA, *Vulkan 1.1.105 - A Specification (with all registered Vulkan extensions*, Aug. 2019. [Online]. Available: `https://www.khronos.org/registry/vulkan/specs/1.1-extensions/pdf/vkspec.pdf`. [Accessed May 9, 2019].

[33] NVIDIA, Santa Clara, CA, USA, *NVIDIA Tesla V100 GPU Architecture*, 2017. [Online]. Available: `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`. [Accessed May 9, 2019].

[34] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," *ACM SIGPLAN Notices*, vol. 52, pp. 31–43, Jan. 2017.

[35] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling Halide image processing pipelines," *ACM Transactions on Graphics*, vol. 35, pp. 83:1–83:11, July 2016.

[36] M. Bauer, S. Treichler, and A. Aiken, "Singe: Leveraging warp specialization for high performance on GPUs," *ACM SIGPLAN Notices*, vol. 49, pp. 119–130, Feb. 2014.

[37] "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 3202–3216, Dec. 2014.

[38] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 79–84, New York, NY, USA: ACM, Mar. 2009.

[39] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, (Piscataway, NJ, USA), pp. 4:1–4:12, IEEE Press, Aug. 2008.

[40] T. Brandvik and G. Pullan, "SBLOCK: A framework for efficient stencil-based PDE solvers on multi-core platforms," in *2010 10th IEEE International Conference on Computer and Information Technology*, (Bradford, United Kingdom), pp. 1181–1188, June 2010.

[41] B. Hamilton and S. Bilbao, "FDTD methods for 3-D room acoustics simulation with high-order accuracy in space and time," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, pp. 2112–2124, Nov. 2017.

[42] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, (New York, NY, USA), pp. 155–164, ACM, Mar. 2012.

[43] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, (New York, NY, USA), pp. 311–320, ACM, June 2012.

[44] J. Pekkilä, M. S. Väisälä, M. Käpylä, P. J. Käpylä, and O. Anjum, "Methods for compressible fluid simulation on GPUs using high-order finite differences," *Computer Physics Communications*, vol. 217, pp. 11–22, Aug. 2017.

[45] R. Collobert, S. Bengio, and J. Marithoz, "Torch: A modular machine learning software library," tech. rep., Idiap Research Institute, Nov. 2002.

[46] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in

Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, (Berkeley, CA, USA), pp. 265–283, USENIX Association, Nov. 2016.

[47] N. J. Goldbaum, H.-Y. Schive, J. A. ZuHone, M. J. Turk, M. Gaspari, and C.-Y. Cheng, "GAMER-2: a GPU-accelerated adaptive mesh refinement code - accuracy, performance, and scalability," *Monthly Notices of the Royal Astronomical Society*, vol. 481, pp. 4815–4840, Sept. 2018.

[48] G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J. hoon Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C. So, F. Zhao, R. Cen, and Y. L. and, "ENZO: An adaptive mesh refinement code for astrophysics," *The Astrophysical Journal Supplement Series*, vol. 211, p. 19, Mar. 2014.

[49] E. E. Schneider and B. E. Robertson, "Cholla: A new massively parallel hydrodynamics code for astrophysical simulation," *The Astrophysical Journal Supplement Series*, vol. 217, p. 24, Apr. 2015.

[50] P. Benítez-Llambay and F. S. Masset, "FARGO3D: A new GPU-oriented MHD code," *The Astrophysical Journal Supplement Series*, vol. 223, p. 11, Mar. 2016.

[51] M. Blazewicz, S. R. Brandt, P. Diener, D. M. Koppelman, K. Kurowski, F. Löffler, E. Schnetter, and J. Tao, "A massive data parallel computational framework for petascale/exascale hybrid computer systems," in *Applications, Tools and Techniques on the Road to Exascale Computing, Proceedings of the conference ParCo 2011, 31 August - 3 September 2011, Ghent, Belgium*, (Ghent, Belgium), pp. 351–358, Aug. 2011.

[52] J. Fung, *A Study of Protoplanetary Disk Dynamics using Accelerated Hydrodynamics Simulations on Graphics Processing Units*. PhD thesis, University of Toronto, Canada, 2015.

[53] C. J. Webb, *Parallel computation techniques for virtual acoustics and physical modelling synthesis*. PhD thesis, University of Edinburgh, Scotland, United Kingdom, 2014.

[54] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The Cactus framework and toolkit: Design and applications," in *Vector and*

Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science, (Berlin, Germany), Springer, Apr. 2003.

[55] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization*, vol. 9, pp. 54:1–54:23, Jan. 2013.

[56] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: A domain specific language for building portable mesh-based PDE solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 9:1–9:12, ACM, Nov. 2011.

[57] M. Steuwer, T. Remmelg, and C. Dubach, "LIFT: A functional data-parallel IR for high-performance GPU code generation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, (Austin, TX, USA), pp. 74–85, Feb. 2017.

[58] M. S. Väisälä, *Magnetic Phenomena of the Interstellar Medium in Theory and Observation*. PhD thesis, University of Helsinki, Finland, 2017.

[59] J. Williamson, "Low-storage Runge-Kutta schemes," *Journal of Computational Physics*, vol. 35, pp. 48–56, Mar. 1980.

[60] D. Luebke and G. Humphreys, "How GPUs work," *Computer*, vol. 40, pp. 96–100, Feb. 2007.

[61] J. Owens, "Streaming architectures and technology trends," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, July 2005.

[62] V. Volkov, *Understanding Latency Hiding on GPUs*. PhD thesis, University of California, CA, USA, 2016.

[63] D. A. Patterson, "Latency lags bandwith," *Communications of the ACM*, vol. 47, pp. 71–75, Oct. 2004.

[64] M. Harris, "Mapping computational concepts to GPUs," in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), ACM, July 2005.

[65] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, (Piscataway, NJ, USA), pp. 1–11, IEEE Press, Nov. 2008.

[66] NVIDIA, Santa Clara, CA, USA, *CUDA C Best Practices Guide*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf`. [Accessed May 9, 2019].

[67] NVIDIA, Santa Clara, CA, USA, *Tuning CUDA Applications for Pascal*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/Pascal_Tuning_Guide.pdf`. [Accessed May 9, 2019].

[68] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug. 2008.

[69] NVIDIA, Santa Clara, CA, USA, *Precision and performance: floating point and IEEE 754 compliance for NVIDIA GPUs*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/Floating_Point_on_NVIDIA_GPU.pdf`. [Accessed May 9, 2019].

[70] AMD, Sunnyvale, CA, USA, *"Vega" Instruction Set Architecture Reference Guide*, July 2017. [Online]. Available: `https://developer.amd.com/wp-content/resources/Vega_Shader_ISA_28July2017.pdf`. [Accessed May 9, 2019].

[71] NVIDIA, Santa Clara, CA, USA, *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 2009. [Online]. Available: `https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`. [Accessed May 9, 2019].

[72] NVIDIA, Santa Clara, CA, USA, *Tuning CUDA Applications for Volta*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/Volta_Tuning_Guide.pdf`. [Accessed May 9, 2019].

[73] NVIDIA, Santa Clara, CA, USA, *Tuning CUDA Applications for Turing*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/Turing_Tuning_Guide.pdf`. [Accessed May 9, 2019].

[74] K. Choo, W. Panlener, and B. Jang, "Understanding and optimizing GPU cache memory performance for compute workloads," in *2014 IEEE 13th International Symposium on Parallel and Distributed Computing*, (Marseilles, France), pp. 189–196, June 2014.

[75] D. Foley and J. Danskin, "Ultra-performance Pascal GPU and NVLink interconnect," *IEEE Micro*, vol. 37, pp. 7–17, Mar. 2017.

[76] The Khronos Group Inc., Beaverton, OR, USA, *The OpenGL Shading Language, Version 4.60.6*, May 2018. [Online]. Available: `https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf`. [Accessed May 9, 2019].

[77] NVIDIA, Santa Clara, CA, USA, *CUDA compiler driver NVCC*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf`. [Accessed May 9, 2019].

[78] NVIDIA, Santa Clara, CA, USA, *CUDA driver API*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_Driver_API.pdf`. [Accessed May 9, 2019].

[79] NVIDIA, Santa Clara, CA, USA, *Parallel Thread Execution ISA Version 6.4*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/ptx_isa_6.4.pdf`. [Accessed May 9, 2019].

[80] NVIDIA, Santa Clara, CA, USA, *CUDA Binary Utilities*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_Binary_Utilities.pdf`. [Accessed May 9, 2019].

[81] Intel, Santa Clara, CA, USA, *UG-20031. High Bandwidth Memory (HBM2) Interface Intel FPGA IP User Guide*, May 2018. [Online]. Available: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-20031.pdf`. [Accessed May 9, 2019].

[82] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," *IEEE Computer Graphics and Applications*, vol. 14, pp. 23–32, July 1994.

[83] P. A. Davidson, *An Introduction to Magnetohydrodynamics*. Cambridge Texts in Applied Mathematics, Cambridge, England: Cambridge University Press, 1st ed., 2001.

[84] A. Emery and H. Mortazavi, "A comparison of the finite difference and finite element methods for heat transfer calculations," in *Proceedings of the NASA/George Washington Univ./Old Dominion Univ. Symposium on Computational Aspects of Heat Transfer*, (Hampton, VA, USA), pp. 51–82, United States, Jan. 1981.

[85] D. I. Ketcheson, "Runge-Kutta methods with minimum storage implementations," *Journal of Computational Physics*, vol. 229, pp. 1763–1773, Mar. 2010.

[86] M. Calvo, J. Franco, J. Montijano, and L. Rández, "On some new low storage implementations of time advancing Runge-Kutta methods," *Journal of Computational and Applied Mathematics*, vol. 236, pp. 3665–3675, Sept. 2012.

[87] D. Zingg and T. Chisholm, "Runge-Kutta methods for linear ordinary differential equations," *Applied Numerical Mathematics*, vol. 31, pp. 227–238, Oct. 1999.

[88] ReSoLVE Centre of Excellence, Espoo, Finland, *Astaroth Repository*, Apr. 2019. [Online]. Available: `https://bitbucket.org/jpekkila/astaroth_2019-05`. [Accessed May 9, 2019].

[89] T. Muranushi, H. Hotta, J. Makino, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, N. Hosono, Y. Maruyama, H. Inoue, H. Yashiro, and Y. Nakamura, "Simulations of below-ground dynamics of fungi: 1.184 PFLOPS attained by automated generation and autotuning of temporal blocking codes," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, (Piscataway, NJ, USA), pp. 3:1–3:11, IEEE Press, Nov. 2016.

[90] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Transactions on Graphics*, vol. 22, pp. 896–907, July 2003.

[91] J. Levine, *Flex & Bison*. Sebastopol, CA, USA: O'Reilly Media, Inc., 1st ed., 2009.

[92] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2006.

[93] B. W. Kernighan, *The C Programming Language*. Upper Saddle River, NJ, USA: Prentice Hall Professional Technical Reference, 2nd ed., 1988.

[94] P. Micikevicius, "GPU performance analysis and optimization," Presentation at the GPU Technology Conference, (Santa Clara, CA, USA), NVIDIA, May 2012.

[95] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, pp. 65–76, Apr. 2009.

[96] M. A. O'Neil and M. Burtscher, "Floating-point data compression at 75 GB/s on a GPU," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, (New York, NY, USA), pp. 7:1–7:7, ACM, Mar. 2011.

[97] Oracle Corporation, Redwood Shores, CA, USA, *Addendum to What Every Computer Scientist Should Know About Floating-Point Arithmetic*, 2015. [Online]. Available: `https://docs.oracle.com/cd/E37069_01/pdf/E39019.pdf`. [Accessed May 9, 2019].

[98] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Trans. Program. Lang. Syst.*, vol. 30, pp. 12:1–12:41, May 2008.

[99] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, pp. 5–48, Mar. 1991.

[100] M. L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001.

[101] N. Higham, *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, 2nd ed., 2002.

[102] J.-M. Muller, "On the definition of ulp(x)," Tech. Rep. RR-5504, INRIA, Feb. 2005.

[103] T. Dombre, U. Frisch, J. M. Greene, M. Hénon, A. Mehr, and A. M. Soward, "Chaotic streamlines in the ABC flows," *Journal of Fluid Mechanics*, vol. 167, pp. 353–391, June 1986.

[104] NVIDIA, Santa Clara, CA, USA, *Profiler User's Guide*, 2019. [Online]. Available: `https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf`. [Accessed May 9, 2019].

[105] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 72–86, Jan. 2017.

[106] C. Lattner, "LLVM: An Infrastructure for Multi-Stage Optimization," Master's thesis, University of Illinois at Urbana-Champaign, IL, USA, 2002.

Appendix A

Tokens of the Astaroth DSL

```
D [0-9]
L [a-zA-Z_\\"]

"Scalar"    { return SCALAR; }              /* Built-in types */
"Vector"    { return VECTOR; }
"Matrix"    { return MATRIX; }
"void"      { return VOID; }                /* Rest of the types inherited from C */
"int"       { return INT; }
"int3"      { return INT3; }

"Kernel"       { return KERNEL; }           /* Function specifiers */
"Preprocessed" { return PREPROCESSED; }

"const"     { return CONSTANT; }
"in"        { return IN; }                  /* Device function storage specifiers */
"out"       { return OUT; }
"uniform"   { return UNIFORM; }

"else if"   { return ELIF; }
"if"        { return IF; }
"else"      { return ELSE; }
"for"       { return FOR; }
"while"     { return WHILE; }

"return"    { return RETURN; }

{D}+"."?{D}*[flud]? { return NUMBER; }      /* Literals */
"."{D}+[flud]?      { return NUMBER; }
{L}({L}|{D})*       { return IDENTIFIER; }
```

```
"=="                { return LEQU; }        /* Logic operators */
"&&"                { return LAND; }
"||"                { return LOR; }
"<="                { return LLEQU; }

"++"                { return INPLACE_INC; }
"--"                { return INPLACE_DEC; }

[-+*/;=\[\]{}(),\.<>] { return yytext[0]; } /* Characters */


"//".*              { /* Skip regular comments */ }
[ \t\n\v\r]+        { /* Ignore whitespace, tabs and newlines */ }
.                   { printf("unrecognized char %d: [%c]\n", *yytext, *yytext); }
```

Appendix B

Grammar of the Astaroth DSL

```
root: program
    ;

program: /* Empty*/
       | program function_definition
       | program assignment ';'        /* Global definition */
       | program declaration ';'       /* Global declaration */
       ;



/*
 * =============================================================================
 * Functions
 * =============================================================================
 */
function_definition: function_declaration compound_statement
                   ;

function_declaration: declaration function_parameter_declaration
                    ;

function_parameter_declaration: '(' ')'
                              | '(' declaration_list ')'
                              ;




/*
```

```
 * ===========================================================================
 * Statements
 * ===========================================================================
 */
statement_list: statement
              | statement_list statement
              ;

compound_statement: '{' '}'
                  | '{' statement_list '}'
                  ;

statement: selection_statement
         | iteration_statement
         | exec_statement ';'
         ;

selection_statement: IF expression else_selection_statement
                   ;

else_selection_statement: compound_statement
                        | compound_statement elif_selection_statement
                        | compound_statement ELSE compound_statement
                        ;

elif_selection_statement: ELIF expression else_selection_statement
                        ;

iteration_statement: WHILE expression compound_statement
                   | FOR for_expression compound_statement
                   ;

for_expression: '(' for_init_param for_other_params ')'
              ;

for_init_param: expression ';'
              | assignment ';'
              ;

for_other_params: expression ';'
                | expression ';' expression
                ;

exec_statement: declaration
```

```
              | assignment
              | expression
              | RETURN return_statement
              ;

assignment: declaration '=' expression
          | expression '=' expression
          ;

return_statement: /* Empty */
                | expression
                ;




/*
 * ====================================================================
 * Declarations
 * ====================================================================
 */
declaration_list: declaration
                | declaration_list ',' declaration
                ;

declaration: type_declaration IDENTIFIER
           | type_declaration array_declaration
           ;

array_declaration: IDENTIFIER '[' ']'
                 | IDENTIFIER '[' expression ']'
                 ;

type_declaration: type_specifier
                | type_qualifier type_specifier
                ;
```

```
/*
 * ========================================================================
 * Expressions
 * ========================================================================
 */
expression_list: expression
               | expression_list ',' expression
               ;

expression: unary_expression
          | expression binary_expression
          ;

binary_expression: binary_operator unary_expression
                 ;

unary_expression: postfix_expression
                | unary_operator postfix_expression
                ;

postfix_expression: primary_expression
                  | postfix_expression '[' expression_list ']'
                  | cast_expression '{' expression_list '}'
                  | postfix_expression '(' ')'
                  | postfix_expression '(' expression_list ')'
                  | type_specifier '(' expression_list ')'
                  | postfix_expression '.' IDENTIFIER
                  ;

cast_expression: /* Empty: implicit cast */
               | '(' type_specifier ')'
               ;

primary_expression: IDENTIFIER
                  | NUMBER
                  | '(' expression ')'
                  ;
```

```
/*
 * ============================================================================
 * Terminals
 * ============================================================================
 */
binary_operator: '+'
               | '-'
               | '/'
               | '*'
               | '<'
               | '>'
               | LEQU
               | LAND
               | LOR
               | LLEQU
               ;

unary_operator: '-'
              | '!'
              | INPLACE_INC
              | INPLACE_DEC
              ;

type_qualifier: KERNEL
              | PREPROCESSED
              | CONSTANT
              | IN
              | OUT
              | UNIFORM
              ;

type_specifier: VOID
              | INT
              | INT3
              | SCALAR
              | VECTOR
              | MATRIX
              ;
```

Appendix C

Solver implementation

```
Preprocessed Scalar
value(in Scalar vertex)
{
    return vertex[vertexIdx];
}

Preprocessed Vector
gradient(in Scalar vertex)
{
    return (Vector){derx(vertexIdx, vertex),
                    dery(vertexIdx, vertex),
                    derz(vertexIdx, vertex)};
}

Preprocessed Matrix
hessian(in Scalar vertex)
{
    Matrix hessian;

    hessian.row[0] = (Vector){derxx(vertexIdx, vertex),
                              derxy(vertexIdx, vertex),
                              derxz(vertexIdx, vertex)};
    hessian.row[1] = (Vector){hessian.row[0].y,
                              deryy(vertexIdx, vertex),
                              deryz(vertexIdx, vertex)};
    hessian.row[2] = (Vector){hessian.row[0].z,
                              hessian.row[1].z,
                              derzz(vertexIdx, vertex)};

    return hessian;
}
```

Listing C.1: Implementation of the stencil assembly stage using the Astaroth domain-specific language.

```
uniform Scalar cs2_sound;
uniform Scalar nu_visc;
uniform Scalar cp_sound;
uniform Scalar cv_sound;
uniform Scalar mu0;
uniform Scalar eta;
uniform Scalar gamma;
uniform Scalar zeta;

uniform int nx_min;
uniform int ny_min;
uniform int nz_min;
uniform int nx;
uniform int ny;
uniform int nz;




Vector
value(in Vector uu)
{
    return (Vector){value(uu.x), value(uu.y), value(uu.z)};
}


Matrix
gradients(in Vector uu)
{
    return (Matrix){gradient(uu.x), gradient(uu.y), gradient(uu.z)};
}


Scalar
continuity(in Vector uu, in Scalar lnrho)
{
    return -dot(value(uu), gradient(lnrho)) - divergence(uu);
}


Vector
induction(in Vector uu, in Vector aa)
{
  const Vector B = curl(aa);
  const Vector grad_div = gradient_of_divergence(aa);
  const Vector lap = laplace_vec(aa);

  return cross(value(uu), B) - eta * (grad_div - lap);
}
```

```
56
57
58 Vector
59 momentum(in Vector uu, in Scalar lnrho, in Scalar ss, in Vector aa)
60 {
61     const Matrix S = stress_tensor(uu);
62     const Scalar cs2 = cs2_sound * exp(gamma * value(ss) / cp_sound
63                                        + (gamma - 1) * (value(lnrho) - LNRHO0));
64     const Vector  j = (Scalar(1.) / mu0) * (gradient_of_divergence(aa)
65                                             - laplace_vec(aa));
66     const Vector B = curl(aa);
67     const Scalar inv_rho = Scalar(1.) / exp(value(lnrho));
68
69     const Vector mom = - mul(gradients(uu), value(uu))
70                        - cs2 * ((Scalar(1.) / cp_sound) * gradient(ss)
71                                 + gradient(lnrho))
72                        + inv_rho * cross(j, B)
73                        + nu_visc * (
74                             laplace_vec(uu)
75                           + Scalar(1. / 3.) * gradient_of_divergence(uu)
76                           + Scalar(2.) * mul(S, gradient(lnrho))
77                           )
78                           + zeta * gradient_of_divergence(uu);
79     return mom;
80 }
81
82
83 Scalar
84 heat_conduction( in Scalar ss, in Scalar lnrho)
85 {
86   const Scalar inv_cp_sound = AcReal(1.) / cp_sound;
87
88   const Vector grad_ln_chi = - gradient(lnrho);
89
90   const Scalar first_term = gamma * inv_cp_sound * laplace(ss)
91                           + (gamma - AcReal(1.)) * laplace(lnrho);
92   const Vector second_term = gamma * inv_cp_sound * gradient(ss)
93                            + (gamma - AcReal(1.)) * gradient(lnrho);
94   const Vector third_term = gamma * (inv_cp_sound * gradient(ss)
95                            + gradient(lnrho)) + grad_ln_chi;
96
97   const Scalar chi = AC_THERMAL_CONDUCTIVITY / (exp(value(lnrho)) * cp_sound);
98   return cp_sound * chi * (first_term + dot(second_term, third_term));
99 }
100
101
102 Scalar
103 lnT( in Scalar ss, in Scalar lnrho)
104 {
105   const Scalar lnT = LNT0 + gamma * value(ss) / cp_sound
106                    + (gamma - Scalar(1.)) * (value(lnrho) - LNRHO0);
107   return lnT;
108 }
109
110
```

```
Scalar
entropy(in Scalar ss, in Vector uu, in Scalar lnrho, in Vector aa)
{
    const Matrix S = stress_tensor(uu);
    const Scalar inv_pT = Scalar(1.) / (exp(value(lnrho))
                                        * exp(lnT(ss, lnrho)));
    const Vector  j = (Scalar(1.) / mu0) * (gradient_of_divergence(aa)
                                            - laplace_vec(aa));
    const Scalar RHS = H_CONST - C_CONST
                   + eta * (AC_mu0) * dot(j, j)
                   + Scalar(2.) * exp(value(lnrho)) * nu_visc * contract(S)
                   + zeta * exp(value(lnrho)) * divergence(uu) * divergence(uu);

    return - dot(value(uu), gradient(ss)) + inv_pT * RHS
           + heat_conduction(ss, lnrho);
}


in Scalar lnrho      = VTXBUF_LNRHO;
out Scalar out_lnrho = VTXBUF_LNRHO;

in Vector uu      = (int3) {VTXBUF_UUX, VTXBUF_UUY, VTXBUF_UUZ};
out Vector out_uu = (int3) {VTXBUF_UUX, VTXBUF_UUY, VTXBUF_UUZ};

in Vector aa      = (int3) {VTXBUF_AX, VTXBUF_AY, VTXBUF_AZ};
out Vector out_aa = (int3) {VTXBUF_AX, VTXBUF_AY, VTXBUF_AZ};

in Scalar ss      = VTXBUF_ENTROPY;
out Scalar out_ss = VTXBUF_ENTROPY;


Kernel void
solve(Scalar dt)
{
    out_lnrho = rk3(out_lnrho, lnrho, continuity(uu, lnrho),       dt);
    out_aa    = rk3(out_aa,    aa,    induction(uu, aa),           dt);
    out_uu    = rk3(out_uu,    uu,    momentum(uu, lnrho, ss, aa), dt);
    out_ss    = rk3(out_ss,    ss,    entropy(ss, uu, lnrho, aa),  dt);
}
```

Listing C.2: Implementation of the stencil processing stage using the Astaroth domain-specific language.

Appendix D

List of Symbols

Symbol	Explanation
$\mathbf{u}$	Velocity
$\mathbf{A}$	Magnetic vector potential
$\mathbf{B}$	Magnetic flux density
$\mathbf{S}$	Traceless rate-of-shear tensor
$\mathbf{f}$	Forcing
ρ	Density
$\gamma = c_p/c_v$	Adiabatic index
ν	Kinematic viscosity
ζ	Bulk viscosity
χ	Thermal diffusivity
μ_0	Magnetic vacuum permeability
η	Magnetic diffusivity
s	Entropy
c_s	Speed of sound
c_p	Heat capacity at constant pressure
c_v	Heat capacity at constant volume
T	Temperature
K	Radiative thermal conductivity
$\mathcal{H}$	Explicit heating term
$\mathcal{C}$	Explicit cooling term
∇^2	Laplace operator
$\nabla\times$	Curl operator

Glossary

cache blocking A technique, where a problem is solved in small subsets, such that the working set at any given time is small enough to fit in caches [3]. 13, 14, 31, 51

data parallelism A form of parallelism, where work is decomposed into data items that can be processed in parallel. 6, 10, 20

error-correcting codes A group of techniques where redundant bits are added to a message to detect and correct errors. 60, 61, 84

graphics pipeline A chain of logical stages executed on graphics primitives to render a scene to a two-dimensional framebuffer. Pipeline in the sense, that output of the previous stage is passed to the next. 11, 20, 21, 45

instruction pipeline A technique used in hardware design to improve instruction-level parallelism. In an instruction pipeline, multiple instructions are executed in parallel, either by dispatching a new instruction before the fetch-decode-execute cycle of the previous instruction has finished, or by issuing multiple instructions to multiple functional units during a single clock cycle [2, 3]. 9, 22

instruction-level parallelism A form of parallelism, where multiple instructions are executed in parallel[2, 3]. 9, 10, 18, 21, 52, 82, 84

latency The time it takes to complete an operation. 11, 13, 18, 21, 22, 25, 27, 52, 72, 74, 75, 78, 82, 84

occupancy The ratio of active warps to the maximum number of warps that can be multithreaded on a SIMT processor [9]. 52, 74, 84, 86

pipeline stall The event, where a processor cannot execute instructions and the instruction pipeline is stalled due to the latency caused by, for example, a data dependency, arithmetic operation, or synchronization. 11, 13, 22, 31, 52, 54, 74, 82, 84

reuse The action of re-accessing data after it has been fetched to on-chip caches. Higher reuse can be achieved by ensuring that data items residing in caches are completely exhausted before evicting them to off-chip memory. 31, 40, 52, 75, 78, 84, 86

stream processor A functional unit specialized in executing common 32- and 64 floating-point operations. 10, 21, 26, 27, 30, 52, 54, 85